Machine learning theory

Vilde Flusgrud

April 15, 2018

1 Introduction

- Estimation (try to determine a model(?) parameter) vs. prediction (try to predict the value of a random variable)
- The review is organized as follows. We begin by introducing polynomial regression as a simple example that highlights many of the core ideas of ML. The next few chapters introduce the language and major concepts needed to make these ideas more precise including tools from statistical learning theory such as overfitting, the bias-variance tradeoff, regularization, and the basics of Bayesian inference. The next chapter builds on these examples to discuss stochastic gradient descent and its generalizations. We then apply these concepts to linear and logistic regression, followed by a detour to discuss how we can combine multiple statistical techniques to improve supervised learning, introducing bagging, boost- ing, random forests, and XG Boost. These ideas, though fairly technical, lie at the root of many of the advances in ML over the last decade. The review continues with a thorough discussion of supervised deep learning and neural networks, as well as convolutional nets. We then turn our focus to unsupervised learning. We start with data visualization and dimensionality reduction before proceeding to a detailed treatment of clustering. Our discussion of clustering naturally leads to an examina- tion of variational methods and their close relationship with mean-field theory. The review continues with a discussion of deep unsupervised learning, focusing on energy-based models, such as Restricted Boltzmann Ma- chines (RBMs) and Deep Boltzmann Machines (DBMs). Then we discuss two new and extremely popular model- ing frameworks for unsupervised learning, generative ad- versarial networks (GANs) and variational autoencoders (VAEs). We conclude the review with an outlook and discussion of promising research directions at the inter- section physics and ML.

2 Why is machine learning difficult?

- Ingredients:
  - $\mathbf{X}$ = the dataset
  - $g(\mathbf{w})$ = the model = a function of the parameters $\mathbf{w}$
  - $\mathcal{C}(\mathbf{X}, g(\mathbf{w}))$ = the cost function, allows us to judge how well the model performs on the observations.
  - $\mathbf{X}_{train}$ = 90% of $\mathbf{X}$
  - $\mathbf{X}_{test}$ = 10% of $\mathbf{X}$
  - $\hat{\mathbf{w}} = argmin_w\{\mathcal{C}(\mathbf{X}_{train}, g(\mathbf{w}))\}$ (the model is fit by minimizing the cost function)
  - $E_{in} = \mathcal{C}(\mathbf{X}_{train}, g(\hat{\mathbf{w}}))$ (the in-sample error)
  - $E_{out} = \mathcal{C}(\mathbf{X}_{test}, g(\hat{\mathbf{w}}))$ (The out-of-sample error. Used to evaluate the performance of the model.)
  - We lmost always find that $E_{in} \geq E_{out}$
- It may be at first surprising that the model that has the lowest out-of-sample error E_{out} usually *does not* have the lowest in-sample error E_{in}. At first glance, the observation that the model providing the best explenation for the current dataset probably will not provide the best explanation for future datasets is very counter-intuitive.

  Moreover, the discrepancy between E_{in} and E_{out} becomes more and more important, as the complexity of our data, and the models we use to make predictions, grows. As the number of parameters in the model increases, we are forced to work in high-dimensional spaces. The "curse of dimensionality" ensures that many phenomena that are absent or rare in low-dimensional spaces become generic. For example, the nature of distance changes in high dimensions, as evidenced in the derivation of the Maxwell distribution in statistical physics where the fact that all the volume of a d-dimensional sphere of radius r is contained in a small spherical shell around r is exploited. Almost all critical points of a function (i.e., the points where all derivatives vanish) are saddles rather than maxima or minima (an observation first made in physics in the context of the p-spin spherical spin glass). For all these reasons, it turns out that for complicated models studied in ML, predicting and fitting are very different things.
- Polynomial regression
  - Probalisitc process that assigns a label y_i to an observation x_i. Data generated by drawing samples from the equation $y_i = f(x_i) + \eta_i$

- $f(x_i)$ = some fixed (but possibly unkown) function. "Function used to generate the data".
- η_i = a Gaussian, uncorrelated noise variable, such that
  - $\langle \eta_i \rangle = 0$
  - $\langle \eta_i \eta_j \rangle = \delta_{ij} \sigma^2$
- σ = the noise strength. ($\sigma = 0$ = noiseless case.)
- $g_\alpha(x; \mathbf{w}_\alpha)$ = a family of functions which depend on some parameters $\mathbf{w}_\alpha$. These functions represent the *model class* that we are using to model the data and make predictions. We chose the model class without knowing $f(x)$. The model class encode the *features* we choose to represent the data. For polynomial regression we will consider three different model classes:
  - i $g_1(x; \mathbf{w}_1)$ = al polynomials of order 1 (two parameters)
  - ii $g_3(x; \mathbf{w}_3)$ = al polynomials up to order 3 (four parameters)
  - iii $g_{10}(x; \mathbf{w}_{10})$ = al polynomials of order 10 (eleven parameters)
- The different number of parameters reflects that the three models have different *model complexities.* Thinking of each term in the polynomial as a "feature", increasing the polynomial order increases the number of features.
- Fit the models on the generated training samples using standard least-squares regression.
- Observe that at small sample sizes, noise can create fluctuations in the data that look like genuine patterns. While simple models are forced to ignore them and focus on the larger trends, complex models can capture both the global trends and noise-generated patterns at the same time. The model can then be tricked into thinking the noise encodes real information = "**overfitting**" = a steep drop-off in predictive performance. Can guard against overfitting in two ways:
  - Use less expressive models with fewer parameters
  - Collect more data so the likelihood that the noise appears patterned decreases.

  This relates to the **bias-variance** tradeoff: when training data limited, one can often get better better predicitve performance by using less expressive model rather than a complex one. The simpler model has more "bias" but is less dependent on the particular realization of the training data., i. e. less "variance". Some universal lessons:
  - Fitting is not predicting. Fitting existing data well is fundamentally different from from making predictions about new data. (fitting = estimation? or not?)

    * Using a complex model can result in overfitting. Increasing a model's complexity will usually yield better results on the training data. However when the training data size is small and the data are noisy, this results in *overfitting* and can substantially degrade the predictive performance of the model.
    * For complex datasets and small training sets, simple models can be better at prediction than complex models due to the bias-variance tradeoff. It takes less data to train a simple model than a complex one. Therefore, even though the correct model is guaranteed to have better predictive performance for an infinite amount of training data (less bias), the training errors stemming from finite-size sampling (variance) can cause simpler models to out-perform the more complex model when sampling is limited.
    * It is difficult to generalize beyond the situations encountered in the training data.

3 Basics of statistical learning theory

- Goal: the sense in which learning is possible, with focus on supervised learning. Ingredients:
  - $y = f(x)$ = an unknown function
  - $\mathcal{H}$ = a hypothesis set that we fix, consisting of all functions we are willing to consider, defined also on the domain of f. The set may be uncountably infinite (e.g. if there are real-valued parameters to fit). Our choices here depends usually on our intuition about the problem.
  - $(x_i, y_i), \quad i = 1...N$ = a set of pairs produced by $f(x)$ which serve as observable data.
  - Our goal: find a function $h \in \mathcal{H}$ approximating $f(x)$ as best as possible, $h \approx f$ in some strict mathematical sense specficed below. Then say we *learned* $f(x)$.
  - If $f(x)$ can in principle take any value on *unobserved* inputs, how is it possible to learn in any meaningful sense? Learning is possible in the restricted sense that the fitted model will probably perform approximately as well on new data as it did on training data.
  - E = appropriately chosen error function (e.g. sum of squared errors in linear regression)
  - When we are training we only have access to E_{in} (fitting). Our goal is to minimize E_{out} (predicting) - the performance on new data.

  - Can we say something about the relationship between E_{in} and E_{out}? Yes: it's the domain of statistical learning theory.

- Three schematics
  - Figure 4: Shows E_{in} and E_{out} as functions of the amount of training data. Assumes large data amount and that the model cannot exactly fit the true function $f(x)$. In the infinite data limit, the two errors must approach the same value, which is our model's **bias** = the best our model could do given infinite training data to beat down sampling noise = a property of the kind of functions/model class we use to approximate $f(x)$. In general: more complex model class = smaller bias. But, do not have infinite data. Thus, better minimize E_{out} than the bias. Can decompose E_{out} into
    * The bias
    * The variance = measures the typical errors introduced in training our model due to sampling noise fraom having a finite training set.

    Final quantity shown is the difference between E_{out} (generalization) and E_{in} (training). Measures difference between fitting and predicting. Models with large difference **overfit** the data. Statistical learning lesson: not enough to minimize E_{in} since E_{out} may still be large. This insight leads to the idea of **regularization**.
  - Figure 5: Shows E_{out} as a function of model complexity = number of parameters/features fex, but not always = model complexity is a subtle idea, defining it precisely is one of the great achievements of statistical learning theory = roughly speaking, it is a measure of the complexity of the model class used to approximate $f(x)$. Considering a training data set of fixed size, E_{out} will be a non-monotonic function of the model complexity and generally minimized for models of *intermediate* complexity. Because: while using a more complex model always reduces the bias, at some point the model becomes too complex for the amount of training data and the generalization error becomes large due to high variance = may be more suitable to use a more biased model with small variance than a less-biased model with large variance = the **bias-variance tradeoff**.
  - Figure 6: Another way to visualize the bias-variance tradeoff. Shows how a complex (high variance, low-bias) model vs a simpler (low variance, high bias) model lands compared to the true model.
    * The complex exhibits larger fluctuations while its average will be closer to the true model. The simpler fluctuates less, but is

on avarage further from the true model. (This reminds one of the accuracy (achieved by the complex model) and precision (achieved by the simpler model) discussion in experimental physics).

    * In general, more complex model needs more training data. This is the cause of the larger fluctuations for the complex model. However, when increasing the data the more complex eventually performs better. Thus, the choice of complexity depends on the amount of training data.

- Bias-Variance Decomposition: dig further into the central principle of the bias-variance tradeoff. Expressiveness vs sensitivity to training data fluctuations. Oftentimes in physics, we are mostly concerned with expressivity, *e.g. whether the true ground state wave function can be well approximated by a class of variational wavefunctions such as a matrix product state.* In the learning context, there is the additional challenge of finding the best variational state with finite sampling. We will see that while this concept is a generally useful heuristic (technique) to keep in mind, it is a mathematically precise statement when decomposing the squared error.(what is meant by this entire sentence?) Finally, we note that a better term would be the bias-variance *decomposition*, as it is possible to have high bias *and* high variance. We'll discuss the b-v tradeoff in the continuous predictions such as regression, but many of the intuitions here also carry over to classification tasks.

  - $\mathcal{L}$ = a dataset
  - $\mathbf{X}_\mathcal{L} = \{(y_j, \mathbf{x}_j), \quad j = 1...N\}$ = the data that makes up $\mathcal{L}$
  - $y = f(\mathbf{x}) + \epsilon$ = a noisy model from which we assume the true data is generated
  - ϵ = normally distributed with mean zero and standard deviation σ_ϵ
  - $\hat{g}_\mathcal{L}(\mathbf{x})$ = a predictor that we assume we have a statistical procedure (e.g. least-squares regression) for forming. The predictor gives the prediction of our model for a new data point $\mathbf{x}$.
  - $\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x})) = \sum_i (y_i - \hat{g}_\mathcal{L}(\mathbf{x}_i))^2$ = the cost function, which we have taken to be the squared error. We choose the estimator previously mentioned estimator by minimizing this.
  - $\{\mathcal{L}_j\}$ = many differen data sets, not just the particular training dataset $\mathcal{L}$ that we have in hand.
  - $E_\mathcal{L}$ = the expectation value of the cost function over $\{\mathcal{L}_j\}$ = the generalization error on all data drawn from the true model

Thus can view $\hat{g}_{\mathcal{L}}$ as a stochastic functional that depends on the dataset $\mathcal{L}$ and can think of $E_{\mathcal{L}}$ as the expected value of the functional if we drew an infinite number of datasets $\{\mathcal{L}_1, \mathcal{L}_2, ...\}$.

- E_ϵ = the expectation value over ϵ, as we would also like to average over different instances over this "noise".

Can thus decompose the expected generalization error in the following way, where in line three we use that $E[X+Y] = E[X] + E[Y]$, in line five use that $Var[X] = E[X^2] - E^2[X]$ and in line six use that our noise has zero mean ($E[\epsilon] = 0$).

$$\begin{aligned}
E_{\mathcal{L},\epsilon}[\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x}))] =& E_{\mathcal{L},\epsilon}[\sum_i (y_i - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] && (1)\\
=& E_{\mathcal{L},\epsilon}[\sum_i (y_i - f(\mathbf{x}_i) + f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] && (2)\\
=& \sum_i E_\epsilon[(y_i - f(\mathbf{x}_i))^2] + E_{\mathcal{L}}[(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] \\
& + 2E_\epsilon[y_i - f(\mathbf{x}_i)]E_{\mathcal{L}}[f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i)] && (3)\\
=& \sum_i E_\epsilon[\epsilon^2] + E_{\mathcal{L}}[(f(\mathbf{x}_i) + 2E_\epsilon[\epsilon]E_{\mathcal{L}}[f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i)] && (4)\\
=& \sum_i \sigma_\epsilon^2 + E_\epsilon^2[\epsilon] + E_{\mathcal{L}}[(f(\mathbf{x}_i) + 2E_\epsilon[\epsilon]E_{\mathcal{L}}[f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i)] && (5)\\
=& \sum_i \sigma_\epsilon^2 + E_{\mathcal{L}}[(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] && (6)
\end{aligned}$$

We further decompose the second term as

$$\begin{aligned}
E_{\mathcal{L}}[(f(\mathbf{x}_i) - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] =& E_{\mathcal{L}}[(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)] + E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)] - \hat{g}_{\mathcal{L}}(\mathbf{x}_i))^2] && (7)\\
=& E_{\mathcal{L}}[(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] \\
& + 2E_{\mathcal{L}}[(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])] && (8)\\
=& (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] \\
& + 2(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)]] && (9)\\
=& (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] \\
& + 2(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 (E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)] - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)]) && (10)\\
=& (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 + E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] && (11)
\end{aligned}$$

where we used that $(f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2$ is a constant ($f(\mathbf{x}_i)$ is deterministic, which means $E[f] = f$ and the expected value of an expected value is just that, $E[E[\hat{g}_{\mathcal{L}}]] = E[\hat{g}_{\mathcal{L}}]$) and that $E[constant] = constant$. Now we have that the two terms we are left with are called

$$Bias^2 = \sum_i (f(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2 \qquad (12)$$

$$Var = \sum_i E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\mathbf{x}_i) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\mathbf{x}_i)])^2] \qquad (13)$$

The bias measures the deviation of the expected value of our estimator (i. e. the asymptotic value of our estimator in the infinite data limit) from the true value. The variance measures how much our estimator fluctuates due to finite-sample effects. Combining we see that the expected out-of-sample error of our model can be decomposed as

$$E_{out} = E_{\mathcal{L},\epsilon}[\mathcal{C}(\mathbf{X}, \hat{g}(\mathbf{x}))] = Bias^2 + Var + Noise \qquad (14)$$

4 Gradient descent and its generalizations

- Ingerdients shared by almost every problem in ML and data science:
  - $\mathbf{X}$ = a dataset
  - $g(\boldsymbol{\theta})$ = a model = a function of the parameters $\boldsymbol{\theta}$
  - $\mathcal{C}(\mathbf{X}, g(\boldsymbol{\theta}))$ = a cost function allowing us to judge how well the model explains the observations. We fit the model by finding the values of $\boldsymbol{\theta}$ that minimize the cost function.

  Here we discuss one of the most powerful and widely used classes of methods for performing this minimization - gradient descent and its generalizations. Main idea: iteratively adjust the parameters in the direction where the gradient of the cost function is large and negative. In this way the training procedure ensures the parameters flow towards a *local* minimum of the cost function.

- Underlying reason training a LM algo is hard is the cost functions we wish to optimize are usually complicated, rugged, non-convex functions in a high-dimensional space with many local minima.

- Gradient descent (GD) and Newton's method:
  - $E(\boldsymbol{\theta})$ = function we wish to minimize. "Energy function". In ML context $E(\boldsymbol{\theta}) = \mathcal{C}(\mathbf{X}, g(\boldsymbol{\theta}))$.
  - Almost always: $E(\boldsymbol{\theta}) = \sum_{i=1}^{n} e_i(\mathbf{x}_i, \boldsymbol{\theta})$ = a sum over n data points. Fex: Linear regression: e_i= the mean square error for data point i. Logistic regression: e_i = the cross-entropy. Call e_i the energy function to make analogy to physical systems.

- Simplest GD: start with an initial value $\boldsymbol{\theta}_0$, then update according to

$$\mathbf{v}_t = \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \tag{15}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{16}$$

  where we have introduced the *learning rate*, η_t, that controls how big a step we should take in the direction of the gradient at time t. For sufficiently small η_t this method will converge to a $localminimum$ of the cost func. But a small η_t comes at a computational cost. If it's smaller we need more steps to reach the minimum. But if it's too large we can overshoot the minimum and the algo becomes unstable (either oscillates or even moves away from the minimum). See fig 7. In practice, one usually specifies a "schedule" that decreases η_t at long times. Common schedules include power law and exponential decays in time.

- Contrast with Newton's method to better understand this behavior of GD. In Newton's method, we choose the step $\mathbf{v}$ for the parameters in such a way as to minimize a second-order Taylor expansion to the energy function

$$E(\boldsymbol{\theta} + \mathbf{v}) \approx E(\boldsymbol{\theta}) + \nabla_\theta E(\boldsymbol{\theta})\mathbf{v} + \frac{1}{2}\mathbf{v}^T H(\theta)\mathbf{v} \tag{17}$$

  where $H(\theta)$ = the Hessian matrix of second derivatives. Differentiating this equation wrt $\mathbf{v}$ and noting that for the optimal value $\mathbf{v}_{opt}$ we expect $\nabla_\theta E(\boldsymbol{\theta} + \mathbf{v}_{opt}) = 0$, yields the equation

$$0 = \nabla_\theta E(\boldsymbol{\theta}) + H(\theta)\mathbf{v}_{opt} \tag{18}$$

  Rearranging results in the desired update rules for Newton's method

$$\mathbf{v}_t = H^{-1}(\theta_t)\nabla_\theta E(\theta_t) \tag{19}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{20}$$

  Have no guarantee the Hessian is well conditioned, so often replaces the Hessian inverse $H^{-1}(\theta_t)$ by some suitably regularized pseudo-inverse such as $[H(\theta_t) + \epsilon I]^{-1}$ w/a small ϵ parameter.

- For ML, Newton's method not practical for two interrelated reasons:
  - Calculating a Hessian is an extremely expensive numerical computation.
  - Even if we employ first order approximation methods to approximate the Hessian (commonly called quasi-Newton methods), we must store and invert a matrix with n^2 entries, n=the number of parameters. For models with millions of parameters such as those commonly employed in neural networks, this is close to impossible with present-day computational power.

- Important intuition from Newton's method to modify GD: Netwon's method automatically adapts the learning rate of different parameters depending on the Hessian matrix. Whereas simple GD has the same learning rate for all the parameters. The Hessian encodes the curvature of the surface we're minimizing. Specifically, the singular values of the Hessian are inversly proportional to the squares of the local curvatures of the surface.

$$\text{the Hessian's singular values} \propto \frac{1}{(\text{the local curvatures of the surface})^2} \tag{21}$$

  Newton's method thus automatically adjusts the step size so that one takes larger steps in flat directions with small curvatures and smaller steps in steep directions with large curvature.

- Consider special case: Using GD to find minimum of a quadratic energy func of a single parameter θ. Given current value of θ, find η_{opt} = the η that lets us reach the minimum in a single step. To find it expand the energy func to second order around the current value

$$E(\theta + v) = E(\theta_c) + \partial_\theta E(\theta) v + \frac{1}{2}\partial_\theta^2 E(\theta) v^2 \tag{22}$$

We want to find the step v such that $\theta + v$ is a stationary point. That is we seek to solve the equation that sets the derivative of this last expression wrt v equal to zero:

$$0 = \partial_\theta E(\theta) + \partial_\theta^2 E(\theta) v \tag{23}$$
$$v = -\partial_\theta E(\theta)[\partial_\theta^2 E(\theta)]^{-1} \tag{24}$$
$$\Rightarrow \theta_{min} = \theta - v \tag{25}$$
$$\theta_{min} = \theta - [\partial_\theta^2 E(\theta)]^{-1}\partial_\theta E(\theta) \tag{26}$$

Comparing with the previously outlined GD update rule tells us that

$$\eta_{opt} = [\partial_\theta^2 E(\theta)]^{-1} \tag{27}$$

Four qualitatively different regimes possible (fig 8):

  - $\eta < \eta_{opt}$: GD will take multiple small steps to reach the bottom of the potential.
  - $\eta = \eta_{opt}$: GD reaches the bottom of the potential in a single step
  - $\eta_{opt} < \eta < 2\eta_{opt}$: GD oscillates across both sides of the potential before eventually converging to the minima.
  - $\eta > 2\eta_{opt}$: GD diverges!

- Straightforward to generalize to the multidimensional case: The natural multidimensional generalization of the second derivative = the Hessian $H(\theta)$. Can always perform a singular value decomposition (= a rotation bt an orthogonal matrix for quadratic minima where the Hessian is symmetric) and consider the Hessian's singular values $\{\lambda\}$. If we use a single learning rate for all parameters, in analogy with the η_{opt} found above, convergence requires that

$$\eta < \frac{2}{\lambda_{max}} \tag{28}$$

where λ_{max}=the Hessian's largest singular value. If the minimum λ_{min} differs significantly from λ_{max}, then convergence in the λ_{min} direction will be extremely slow. Can show that convergence time scales w/ the condition number

$$\kappa = \frac{\lambda_{max}}{\lambda_{min}} \tag{29}$$

- Limitations of the simplest GD algo:
  - *GD finds local minima* Since GD is deterministic it converges to a local minimum of our energy func. May lead to poor performance in ML. A similar problem is encountered in physics and overcome by methods like **simulated annealing** that introduce a fictitious "temperature" which is eventually taken to zero. The "temperature" term introduces stochasticity in the form of thermal fluctuations that allow the algo to thermally tunnel over energy barriers. Suggests we should in ML modify GD to include stochasticity.
  - *GD is sensitive to initial conditions* Initial conditions matter, as a consequence of GD's local nature. Thus, very important to think about how you initialize, both for simple and more complicated GD variant introduced later.
  - *Gradients are computationally expensive to calculate for large datasets* As previously mentioned, in many statistics and ML cases, the energy func is a sum of terms, one for each data point = to calc the gradient we must sum over all n data points. Doint this at each GD point = extremely computationally expensive. An ingenious solution: calc the gradients using small subsets of data = "mini batches". This also introduces stochasticity into our algo.
  - *GD very sensitive to choices of learning rates* Small lr = slow, large lr = possible divergence/poor results. Also, depending on the local landscape, we have to modify lr to ensure convergence. Ideally would "adaptively" choose lr to match the landscape.
  - *GD treats all directions in parameter space uniformly* Unlike Netwon's method. Thus the maximum lr is set by the behavior of the steepest direction and this can significantly slow training. Would ideally like large steps in flat dir and small steps in steep dir. Since we are exploring rugged landscapes where curvatures change, this requires us to keep track of not only the gradient but second derivatives of the energy func (as discussed calc Hessian would be ideal, but proves too computationally expensive).
  - *GD can take exponential time to escape saddle points, even with random initialization* As mentioned extremely sensitive to initial conditions since it determines the particular local minimum GD will reach. But, even with a good initialization scheme (through the introduction of randomness) GD can still take exponential time to escape saddle points, prevalent in high-dimensional space, even for non-pathological objective functions. **There are modified GD methods developed recently to accelerate the escape, see reference**.

- Stochastic gradient descent (SGD) with mini-batches: Stochasticity is incorporated by approximating the gradient on a subset of the data called a mini-batch.
  - Size of mini-batch almost always $\ll$ total number of data points n.
  - Typical mini-batch sizes ranging from ten to a few hundred data points
  - If there are n points, and the mini-batch size is M, there will be n/M mini-batches.
  - Denote these mini-batches by B_k where $k = 1...n/M$.

  Thus in SGD at each GD step we approximate the gradient using a minibatch B_k,

  $$\nabla_\theta = \sum_i^n \nabla_\theta e_i(\mathbf{x}_i, \boldsymbol{\theta}) \rightarrow \sum_{i \in B_k} \nabla_\theta e_i(\mathbf{x}_i, \boldsymbol{\theta}) \tag{30}$$

  cycling over all M minibatches. A full iteration over all n data points = using all M minibatches= an *epoch*. Denote the minibatch approximation to the gradient by

  $$\nabla_\theta E^{MB}(\boldsymbol{\theta}) = \sum_{i \in B_k}^M \nabla_\theta e_i(\mathbf{x}_i, \boldsymbol{\theta}) \tag{31}$$

  Then SGD algo is

  $$\mathbf{v}_t = \eta_t \nabla_\theta E^{MB}(\boldsymbol{\theta}) \tag{32}$$
  $$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{33}$$

  Two important benefits to SGD.

  - Introduces stochasticity and decreases chance the fitting algo gets stuck in isolated local minima.
  - Significantly speeds up the calc as one does not have to use all n data points to calc the gradient.
  - Empirical and theoretical work suggets SGD has additional benefits - one significant being that introducing stochasticity is thought to act as **a natural regularizer that prevents overfitting** in deep, isolated minima.

- Adding momentum: SGD almost always used with a "momentum"/inertia term serving as a memory of the direction we are moving in parameter space. Implemented as

  $$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \tag{34}$$
  $$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_t \tag{35}$$

where γ=a momentum parameter with $0 \leq \gamma \leq 1$ (and have dropped the explicit notation indicating the gradient is taken over a minibatch). This is gradient descent with momentum (GDM). Clear that

- $\mathbf{v}_t$ is a running average
- $(1-\gamma)^{-1}$ sets the characteristic time scale for the memory used in the averaging procedure.
- $\gamma = 0$ reduces down to ordinary SGD

Equivalent way of writing updates:

$$\Delta\boldsymbol{\theta}_{t+1} = \gamma\Delta\boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \tag{36}$$

where $\Delta\boldsymbol{\theta}_t = \boldsymbol{\theta}_t\boldsymbol{\theta}_{t-1}$.

Getting intuition: Consider simple physical analogy with a particle of mass m moving in a viscous medium with drag coefficient μ and potential $E(\mathbf{w})$, $\mathbf{w}$=the particle's position. It's motion then described by

$$m\frac{d^2\mathbf{w}}{dt^2} + \mu\frac{d\mathbf{w}}{dt} = -\nabla_w E(\mathbf{w}) \tag{37}$$

Discretize in usual way to get

$$m\frac{\mathbf{w}_{t+\Delta t} - 2\mathbf{w}_t + \mathbf{w}_{t-\Delta t}}{(\Delta t)^2} + \mu\frac{\mathbf{w}_{t+\Delta t} - \mathbf{w}_t}{\Delta t} = -\nabla_w E(\mathbf{w}) \tag{38}$$

Rearrange to write as

$$\Delta\mathbf{w}_{t+\Delta t} = -\frac{(\Delta t)^2}{m+\mu\Delta t}\nabla_w E(\mathbf{w}) + \frac{m}{m+\mu\Delta t}\Delta\mathbf{w}_t \tag{39}$$

Notice it's identical to our GDM update rule defined above. We may thus identify the momentum parameter and learning rate with the mass of the particle and the viscous drag:

$$\gamma = \frac{m}{m+\mu\Delta t} \tag{40}$$

$$\eta = \frac{(\Delta t)^2}{m+\mu\Delta t} \tag{41}$$

Thus as suggested by the name the momentum parameter is proportional to the mass of the particle and effectively provides inertia. Also, in the large viscosity/small learning rate limit, our memory scales as $(1-\gamma)^{-1} \approx m/(\mu\Delta t)$

Why momentum useful?

  - Helps the algo gain speed in diretions with persistent but small gradients even in the presence of stochasticity, while surpressing oscillations in high-curvature directions.
  - Has been argued first-order methods (with appropriate initial conditions) can perform comparable to more expensive second-order methods, especially in context of complex deep learning models (reference)
  - Studies suggest benefits of momentum especially pronounced in complex models in the initial "transient phase" of training, rather than during subsequent fine-tuning of a coarse minimum. Because in the transient phase, correlations in the gradient persist across many GD steps, accentuating the role of inertia and memory.

  These can be even more pronounced using a slightly modified algo, Nesterov Accelerated Gradient (NAG). Rather than calc gradient at the current parameters $\nabla_\theta E(\boldsymbol{\theta}_t)$, calc the gradient at the expected value of the parameters given our current momentum $\nabla_\theta E(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1})$. One major advantage is it allows for larger learning rate than GDM for same choice of γ.

- Methods that use the second moment of the gradient: In SGD, with and without momentum, we still have to specify a "schedule" for tuning the learning rates η_t as a func of time. As touched upon before this presents dilemmas.
  - The lr is limited by the steepest direction which can change depending on the current position in the landscape. To circumvent this, ideally our algo would keep track of curvature and take large steps in shallow/flat dirs and small steps in steep/narrow dirs.
  - Second-order methods accomplish this by calc or approximating the Hessian and normalizing the lr by the curvature.
  - But this is very computationally expensive for extremely large models.
  - Ideally, we would like to adaptively change the step size to match the landscape without paying the steep computational price of calculating/approximating the Hessian.
  - Recently introduced a number of methods that accomplish this by tracking not only the gradient, but also the second moment of the gradient. These include AdaGrad, AdaDelta, RMS-Prop, and ADAM. We'll discuss the last two.

  RMS prop: In addition to keeping a running average of the first moment of the gradient, also keep track of the second moment denoted by $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. Update rule is

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \tag{42}$$

$$\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \tag{43}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}} \tag{44}$$

where

- β = controls the averaging time of the second moment and is typically $\beta = 0.9$
- η_t = a learning rate, typically 10^{-3}
- $\epsilon \sim 10^{-8}$ = a small regularization constant to prevent divergences
- Multiplication and division by vectors is understood as element-wise operations
- Clear from the formula that lr reduced in dirs where the norm of the gradient is consistently large. This greatly speeds up the convergence by allowing us to use a larger lr for flat dirs.

The ADAM optimizer: Keep a running average of both the first and second moment of the gradient ($\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t]$ and $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$ respectively) - use this info to adaptively change the lr for different parameters. Also performs an additional bias correction to account for the fact that we're estimating the first two moments of the gradient using a running average (denoted below by the hats). Update rule is (multiplication and division by vectors again understood to be element-wise):

$$\mathbf{g}_t = \nabla_\theta E(\boldsymbol{\theta}) \tag{45}$$

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \tag{46}$$

$$\mathbf{s}_t = \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \tag{47}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \tag{48}$$

$$\hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t} \tag{49}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon} \tag{50}$$

where

- β_1 and β_2 set the memory lifetime of the first and second moment, typically taken as 0.9 and 0.99 respectively
- η and ϵ identical to RMS prop.

Like in RMSprop the effective step size of a parameter depends on the magnitude of its gradient squared. To better understand, let's rewrite this expression in terms of the variance $\sigma_t^2 = \hat{\mathbf{s}}_t - (\hat{\mathbf{m}}_t)^2$. Consider a single parameter θ_t. Update rule of it is given by

$$\Delta\theta_{t+1} = -\eta_t \frac{\hat{\mathbf{m}}_t}{\sqrt{\sigma_t^2 + \hat{\mathbf{s}}_t^2} + \epsilon} \tag{51}$$

We examine the limiting cases of this expression.

- Assume our gradient estimates are consistent so the variance is small. Then update rule tends to $\Delta\theta_{t+1} \to -\eta_t$ (assumed $\hat{\mathbf{m}}_t \gg \epsilon$). This is equivalent to cutting off large persistent gradients at 1 and limiting the max step size in steep directions.
- Imagine the gradient is widely fluctuating between GD steps. Then $\sigma^2 \gg \hat{\mathbf{m}}_t^2$ so our update becomes $\Delta\theta_{t+1} \to -\eta_t \hat{\mathbf{m}}_t/\sigma_t$. AKA, we adapt our lr so that

  learning rate $\propto$ signal-to-noise ratio (i.e. the mean in units of the standard deviation)

- = extremely desirable from physical point of view. The std serves as a natural adaptive scale for deciding whether a gradient is large or small.
- Thus, ADAM has the beneficial effects of adapting our step size so that we cut off large gradient dirs (and hence prevent oscillations and divergences) and measuring gradients in terms of a natural length scale, the std σ_t.
- Above discussion also explains empirical observations showing that the performance of both ADAM and RMSprop is drastically reduced if the square root omitted in the update rule.
- Also worth noting recent studies have shown adaptive methods like RMSprop, ADAM, and AdaGrad tend to generalize worse than SGD in classification tasks, though they achieve smaller training error. See refs.

• Comparisons of various methods: Visualize the performance of the 5 discussed methods GD, GDM, NAG, ADAM and RMSprop by using Beale's function:

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \tag{52}$$

It has a global minimum at $(x, y) = (3, 0.5)$ and an interesting structure. Fig 9:

- Shows the results of using all five methods for $N_{steps} = 10^4$ steps from three different initial conditions.
- GD, GDM, NAG learning rate: $\eta = 10^{-6}$

  - RMSprop, ADAM learning rate: $\eta = 10^{-3}$ (can be higher due to their adaptive step sizes. thus these methods tend to be quicker at navigating the landscape.)
  - Notice in some cases (e.g. initial condition (-1, 4)), the trajectories do not find the global min but instead follow the deep, narrow ravine that occurs along $y = 1$. This kind of landscape structure is generic in high-dimensional spaces where saddle points proliferate.

- Gradient descent in practice: practical tips (especially in the context of deep neural networks):

  - *Randomize the data when making mini-batches* Important to randomly shuffle the data when forming the batches. Otherwise GD can fit spurious correlations resulting from the order in which data is presented.
  - *Transform your inputs* As discussed learning=difficult when landscape has miz of steep and flat dirs. Trick: standardize the data by subtracting the mean and normalizing the variance of input variables. Whenever possible also decorrelate the inputs. Why helpful - consider the case of linear regression: for the squared error cost func, the Hessian of the energy matrix = the correlation matrix between inputs. = by standardizing the inputs we ensure the landscape looks homogeneous in all dirs in parameter space. Most deep networks can be viewed as linear transformations followed by a non-linearity at each layer, thus we expect this intuition to hold beyond the linear case.
  - *Monitor the out-of-sample performance* Always monitor the performance on a validation set = a small portion of the training data kept out of the training process to serve as a proxy for the test set. Validation error starting to increase = model being overfit. Terminate the learning. This *early stopping* significantly improves performance in many settings.
  - *Adaptive optimization methods don't always have good generalization* As mentioned recent studies have shown adaptive methods such as ADAM, RMSprop and AdaGrad to have poor generalization campared to SGD or GDM, particularly in the high-dimensional limit = the number of parameters exceeds the number of data points. Although not clear why these methods perform so well in training deep neural networks such as generative adversarial networks (GANs), simpler procedures like properly tuned SGD may work as well or better in these applications.

5 Overview of Bayesian inference

Statistical modeling focus: estimation/prediction of unknown quantities. Bayesian methods premise: probability can be used as mathematical tool for describing uncertainty. Similar in spirit to physics statistical mechanics - where we use probability to describe the behavior of large systems where we cannot know the positions and momenta of all particles even if the system itself is fully deterministic (at least classically). This section gives introduction to Bayesian inference, w/special emphasis on its logic=Bayesian reasoning and connections to ML.

5.1 Bayes rule

Must specify two functions

- $p(\mathbf{X}|\mathbf{w})$ = the likelihood function, which describes the probability of observing a dataset $\mathbf{X}$ for a given value of the unknown parameters $\mathbf{w}$. The func should be considered a func of the parameters $\mathbf{w}$ with the data $\mathbf{X}$ held fixed.
- $p(\mathbf{w})$ = the prior distribution, which describes any knowledge we have about the parameters before we collect the data.
- with these two we can compute the posterior distribution via Baye's rule:

$$p(\mathbf{w}|\mathbf{X}) = \frac{p(\mathbf{X}|\mathbf{w})p(\mathbf{w})}{\int d\mathbf{w}p(\mathbf{X}|\mathbf{w})p(\mathbf{w})} \tag{53}$$

  It describes our knowledge about the unknown parameter $\mathbf{w}$ after observing the data $\mathbf{X}$. In many casing computing the normalizing constant (i.e. the partition function $p(\mathbf{X}) = \int d\mathbf{w}p(\mathbf{X}|\mathbf{w})p(\mathbf{w})$) and Markov Chain Monte Carlo (MCMC) methods are needed to draw random samples from the posterior dist.

The likelihood func = common feature in both classical statistics and Bayesian inference. Determined by the model and the measurement noise. Many statistical procedures fex least-square fitting can be cast into the formalism of *Maximum Likelihood Estimation* (MLE). In MLE one chooses the parameters $\hat{\mathbf{w}}$ that maximize the likelihood (or equivalently the log-likelihood since log a monotonic func) of the observed data:

$$\hat{\mathbf{w}} = argmax_w\{log(p(\mathbf{X}|\mathbf{w}))\} \tag{54}$$

Aka, in MLE we choose the parameters that maximize the probability of seeing the observed data given our generative model. MLE is an important concept in both frequentist and Bayesian statistics.

The prior distribution, by contrast, is uniquely Bayesian. Two general classes of priors:

- If we do not have any specialized knowledge of $\mathbf{w}$ before we look at the data we would like to select an *uninformative* prior that reflects our ignorance
- Otherwise we select an *informative* prior that accurately reflects the knowledge we have about $\mathbf{w}$.

There is a large literative on uninformative priors, including reparametrization invariant priors that would be of interest to physicists, but here we'll focus on *informative* priors. Using this tends to decrease the variance of the posterior distribution while, potentially, increasing its bias. Beneficial if decrease in variance $\gg$ increase in bias.

In high-dimensional problems it's reasonable to assume many of the parameters not strongly relevant $=$ many of the parameters will be zero or close to zero. Can express this belief using two commonly used priors:

- $p(\mathbf{w}|\lambda) = \prod_j \sqrt{\frac{\lambda}{2\pi}} e^{-\lambda w_j^2}$ $=$ the Gaussian prior $=$ used to express the assumption many of the parameters will be small
- $p(\mathbf{w}|\lambda) = \prod_j \frac{\lambda}{2} e^{-\lambda |w_j|}$ $=$ the Laplace prior $=$ used to express the assumption that many of the parameters will be zero.

Come back to this in section VI.F.

5.2 Bayesian decisions

We have seen how to compute the posterior distribution, which expresses our knowledge about the parameters $\mathbf{w}$. Mostly however we need to summarize our knowledge and pick a single "best" value for them. In princple the specific value should be chosen to maximize a utility function. In practice however, usually use one of two choices:

- $\langle \mathbf{w} \rangle = \int d\mathbf{w} \mathbf{w} p(\mathbf{w}|\mathbf{X})$ $=$ the posterior mean $=$ the Bayes estimate (minimizes the mean-squared error)
- $\hat{\mathbf{w}} = argmax_w p(\mathbf{w}|\mathbf{X})$ $=$ the posterior mode $=$ the maximum-a-posteriori $=$ MAP estimate (easier to compute)

5.3 Hyperparameters

The Gaussian and Laplace prior dist. both have an extra parameter $\lambda =$ *hyperparameter* $=$ *nuisance variable.* Has to be chosen somehow.

- One standard approach: define another prior dist for λ, usually using an uninformative prior, and average the posterior dist over all choices of λ. $=$ a hierarchical prior. However computing averages often requires long MCMC simulations that are computationally expensive.

- Simpler: find a good λ value using an optimization procedure. Will discuss later.

6 Linear regression

In this section we take a closer look at the ideas of the optimal choice of predictor depending on the choice of fitted function and underlying noise level, and model complexity, the bias-variance decomposition, the statistical meaning of learning. As previously fitting a given set of samples $(y_i, \mathbf{x}_i)$ means relating the independent variables $\mathbf{x}_i$ to their responses y_i.

Formulating the problem:

- $\mathcal{D} = \{(y_i, \mathbf{x}_i)\}_{i=1}^n$ = a given dataset with n samples, where $\mathbf{x}_i$ = the i-th observation vector while y_i = its corresponding (scalar) response.
- $\mathbf{x}_i \in \mathbb{R}^p$ = assume every sample has p features.
- f = the true function/model that generated these samples via $y_i = f(\mathbf{x}_i; \mathbf{w}_{true}) + \epsilon_i = \mathbf{x}_i^T \mathbf{w}_{true} + \epsilon_i$ for some unknown but fixed $\mathbf{w}_{true} \in \mathbb{R}^p$.
- g = the function we wish to find, with parameters $\mathbf{w}$ fit to the data $\mathcal{D}$, that can best approximate f. When we have a $\hat{\mathbf{w}}$ such that $g(\mathbf{x}; \hat{\mathbf{w}})$ yields our best estimate of f, we can use this g to make predictions about the response y_0 for a new data point $\mathbf{x}_0$.
- L^p for any real number $p \geq 1$ = the norm of a vector $\mathbf{x} = (x_1, ..., x_d) \in \mathbb{R}^d$, defined as

$$||\mathbf{x}||_p = (|x_1|^p + ... + |x_d|^p)^{\frac{1}{p}} \tag{55}$$

6.1 Least-square regression

Ordinary least squares linear regression (OLS) = the minimization of the L_2 norm of the difference between the response y_i and the predictor $g(\mathbf{x}_i; \mathbf{w}) = \mathbf{x}_i^T \mathbf{w}$:

$$min_{\mathbf{w} \in \mathbb{R}^p} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 = min_{\mathbf{w} \in \mathbb{R}^p} \sum_{i=1}^n (\mathbf{x}_i^T \mathbf{w} - y_i)^2 \tag{56}$$

Geometrically, the predictor func g defines a hyperplane in $\mathbb{R}^p$. Thus minimizing least-square error = minimizing the sum of all projections (=residuals) for all points $\mathbf{x}_i$ to this hyperplane (fig 10). Denote the solution to this as $\hat{\mathbf{w}}_{LS}$:

$$\hat{\mathbf{w}}_{LS} = argmin_{\mathbf{w} \in \mathbb{R}^p} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 \tag{57}$$

which after differentiation leads to

$$\hat{\mathbf{w}}_{LS} = (X^T X)^{-1} X^T \mathbf{y} \tag{58}$$

where we assumed $X^T X$ is invertible.

- Often the case when $n \gg p$. Formally: if $rank(X) = p$, namely, the predictors $X_1, ..., X_p$ (=the columns of X) are linearly independent, then $\hat{\mathbf{w}}_{LS}$ is unique.
- $rank(X) < p$: happens when $p > n$. $X^T X$ singular. Implying there are infinitely many solutions to the least squares problem. Can then show that if $\mathbf{w}_0$ a solution, then also $\mathbf{w}_0 + \eta$ also a solution for any η which satisfies $\mathbf{X}\eta = 0$ ($= \eta \in null(X)$).

Having the solution, we can calculate $\mathbf{y}$, the best fit of our data X, as

$$\hat{\mathbf{y}} = X\hat{\mathbf{w}}_{LS} = P_X \mathbf{y} \tag{59}$$

$$\text{where} \quad P_X = X(X^T X)^{-1} X^T \tag{60}$$

Geometrically, P_X = the projection matrix which acts on $\mathbf{y}$ and projects it onto the column space of X, which is spanned by the predictors $X_1, ..., X_p$ (fig 11).

Notice: we found the optimal solution $\hat{\mathbf{w}}_{LS}$ in one shot, no iterative optimization.

Have explained the difference between *learning* and *fitting* lies in the prediction on "unseen" data. Must thus examine out-of-sample error. Following our previos definitions in section 3 of $\bar{E}_{in}$ and $\bar{E}_{out}$, the average in-sample and out-of-sample error can be shown to be

$$\bar{E}_{in} = \sigma^2 (1 - \frac{p}{n}) \tag{61}$$

$$\bar{E}_{out} = \sigma^2 (1 + \frac{p}{n}) \tag{62}$$

provided we obtain $\hat{\mathbf{w}}_{LS}$ from i.i.d. samples X and $\mathbf{y}$ generated through $\mathbf{y} = X\mathbf{w}_{true} + \epsilon$ (this requires ϵ is a noise vector whose elements are i.i.d. of 0 mean nad variance σ^2, and is independent of the samples X). Can thus calc the average generalization error explicitly:

$$|\bar{E}_{in} - \bar{E}_{out}| = 2\sigma^2 \frac{p}{n} \tag{63}$$

This imparts important message:

- $p \gg n$ (= high dimensional data): the generalization errror is extremely large = the model is not learning
- $p \approx n$: even now we might still not learn well due to the intrinsic noise σ^2. One way of amelioration = reularization. Mainly two forms:
  - *Ridge regression*: employs an L_2 penalty
  - *LASSO*: uses an L_1 penalty

6.2 Ridge-regression

We here study effect of adding to the least squares loss function a *regularizer* defined as the L_2 norm of the parameter vector we want to optimize over = wish to solve the following *penalized* regression problem, *Ridge regression*:

$$\hat{\mathbf{w}}_{Ridge}(\lambda) = argmin_{\mathbf{w}\in\mathbb{R}^p}(||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_2^2) \tag{64}$$

Equivalent to the following *constrained* optimization problem

$$\hat{\mathbf{w}}_{Ridge}(t) = argmin_{\mathbf{w}\in\mathbb{R}^p:||\mathbf{w}||_2^2\leq t}||X\mathbf{w} - \mathbf{y}||_2^2 \tag{65}$$

Means for any $t \geq 0$ and $\hat{\mathbf{w}}_{Ridge}$ in the last equation there exists a $\lambda \geq 0$ such that $\hat{\mathbf{w}}_{Ridge}$ solves the second last equation, and vice versa (note: equivalence between the penalized and the constrained (regularized) form of least square does not always hold. It holds for Ridge and LASSO, but not for best subset selection defined by choosing a L^0 norm).

With this equivalence it's obvious that by adding a regularization term $\lambda||\mathbf{w}||_2^2$ to our ls loss func, we are effectively constraining the magnitude of the parameter vector learned from the data. To see this, let's evaluate the second last equation explicitly. Differentiating w.r.t $\mathbf{w}$ gives

$$\hat{\mathbf{w}}_{Ridge}(\lambda) = (X^TX + \lambda I_{p\times p})^{-1}X^T\mathbf{y} \tag{66}$$

$$\text{If X orthogonal:} \quad = \frac{\hat{\mathbf{w}}_{LS}}{1+\lambda} \tag{67}$$

Implies the ridge estimate is merely the least squares estimate scaled by a factor $(1+\lambda)^{-1}$.

Can we derive similar relation between the fitted vector $\hat{\mathbf{y}} = X\hat{\mathbf{w}}_{Ridge}$ and the prediction made by ls linear regression? We do a singular value decomposition (SVD) on X. The SVD of an $n \times p$ matrix X has the form

$$X = UDV^T \tag{68}$$

where $[U]_{n\times p}$ and $[V]_{p\times p}$ are orthogonal matrices such that the columns of U span the column space of X while the columns of V span the row space of X. $[D]_{p\times p} = diag(d_1, d_2, ..., d_p)$ is a diagonal matrix with entries $d_1 \geq d_2 \geq ...d_p \geq 0$ called the singular values of X. X is singular if there is at least one $d_j = 0$. By writing X in terms of its SVD, one can recast the Ridge estimator as

$$\hat{\mathbf{w}}_{Ridge} = V(D^2 + \lambda I)^{-1}DU^T\mathbf{y} \tag{69}$$

which implies that the Ridge predictor satisfies

$$
\begin{aligned}
\hat{\mathbf{y}}_{Ridge} =& X\hat{\mathbf{w}}_{Ridge} && (70)\\
=& UD(D^2+\lambda I)^{-1}DU^T\mathbf{y} && (71)\\
=& \sum_{j=1}^{p} \mathbf{u}_j \frac{d_j^2}{d_j^2+\lambda}\mathbf{u}_j^T\mathbf{y} && (72)\\
\leq& UU^T\mathbf{y} && (73)\\
=& X\hat{\mathbf{y}} \equiv \hat{\mathbf{y}}_{LS} && (74)
\end{aligned}
$$

where in the inequality step we used SVD to simplify $\hat{\mathbf{w}}_{LS}$ and asumed $\lambda \geq 0$. Comparing the third last line and last line it's clear that

- In order to compute the fitted vector $\hat{\mathbf{y}}$, both Ridge and ls linear regression have to project $\mathbf{y}$ to the column space of X.
- Only difference: Ridge regression further shrinks each basis component j by a factor $d_j^2/(d_j^2+\lambda)$

6.3 LASSO and sparse regression

Here study the effects of adding an L_1 regularization penalty, called *LASSO* (=least absolute shrinkage and selection operator). LASSO in the penalized form is defined by the following regularized regression problem:

$$\hat{\mathbf{w}}_{LASSO}(\lambda) = argmin_{\mathbf{w}\in\mathbb{R}^p}||X\mathbf{w}-\mathbf{y}||_2^2 + \lambda||\mathbf{w}||_1 \tag{75}$$

As with Ridge there is another formulation based on constrained optimization:

$$\hat{\mathbf{w}}_{LASSO}(t) = argmin_{\mathbf{w}\in\mathbb{R}^p:||\mathbf{w}||_1\leq t}||X\mathbf{w}-\mathbf{y}||_2^2 \tag{76}$$

- The equivalence interpretation same as in Ridge, namely: for any $t \geq 0$ and solution $\hat{\mathbf{w}}_{LASSO}$ in the last equation there is a $\lambda \geq 0$ such that $\hat{\mathbf{w}}_{LASSO}$ solves the second last equation. And vice versa.
- However, to get the analytical solution of LASSO cannot simply take gradient wrt $\mathbf{w}$, since the L_1-regularizer is not everywhere differentiable, in particular at any point where $w_j = 0$ (fig 13).
- Nonetheless, LASSO is a **convex** problem. Can thus invoke the *subgradient optimality condition* in optimization theory to obtain solution.

For simple notation, only show the solution for when X is orthogonal:

$$\text{X orthogonal:} \quad \hat{w}_j^{LASSO}(\lambda) = sign(\hat{w}_j^{LS})(|\hat{w}_j^{LS}|-\lambda)_+ \tag{77}$$

where $(x)_+ =$ the positive part of x.

Fig 12 compares the Ridge and LASSO solutions.

- As mentioned, Ridge is the LS solution scaled by a factor $(1+\lambda)$. LASSO does something called "soft-thresholding".
- LASSO tends to give sparse solutions, meaning many components of $\hat{\mathbf{w}}_{LASSO}$ are zero. Fig 13 gives intuitive explanation of this. In short:
  - To solve a constrained optimization problem with a fixed regularization strength $t \geq 0$ one first carves out the "feasible region" specified by the regularizer in the $\{w_1, ..., w_d\}$ space = a solution $\hat{\mathbf{w}}_0$ only legitimate if falls within this region.
  - Then one plots the contours of the ls regressors in an increasing manner until the contour touches the feasible region.
  - The point where this occurs is the solution to our optimization problem.
  - Loosely speaking, since the L_1 regularizer of LASSO has sharp protrusions(=vertices) along the axes, and because the regressor contours are in the shape of ovals (it's quadratic in $\mathbf{w}$), their intersection tends to occur at the vertex of the feasibility region = implying the solution vector will be sparse. (the vertices correspond to parameter vectors $\mathbf{w}$ with only one non-vanishing component).

See fig 14 and 15 for comparison of Ridge and Lasso on Diabetes dataset.

6.4 Using linear regression to learn the Ising Hamiltonian

Task: learning the Hamiltonian for the Ising model. Given an ensamble of random spin configurations, and assigned to each state its energy, generated from the 1D Ising model:

$$H = -J \sum_{j=1}^{L} S_j S_{j+1} \tag{78}$$

where

- J = the nearest-neighbour spin interaction
- $S_j \in \{\pm 1\}$ = a spin variable
- Assume data was generated with $J = 1$
- You are handed the dataset $\mathcal{D} = (\{S_j\}_{j=1}^{L}, E_j)$ without knowing what the E_j number mean
- The $\{S_j\}_{j=1}^{L}$ cofiguration can be interpreted in many ways: outcome of coin tosses, black-and-white image pixels, binary representation of integers, etc.

- Goal: learn a model that predicts E_j from the spin configs.

Solving the problem with linear regression (lr):

- Without any prior knowledge of the dataset's origin, physics intuition may suggest: look for a spin model w/pairwise interactions between every pair of variables. That is we choose the following model class:

$$H_{model}[S^i] = -\sum_{j=1}^{L}\sum_{k=1}^{L} J_{j,k} S_j^i S_k^i \tag{79}$$

- Goal: Determine the interaction matrix $J_{j,k}$ by applying linear regression on the dataset $\mathcal{D}$ = a well defined problem, since the unknown $J_{j,k}$ enters linearly into the definition of the Hamiltonian.
- To this end we cast the above ansatz into the more familiar linear regression form:

$$H_{model}[S^i] = \mathbf{X}^i \cdot \mathbf{J} \tag{80}$$

- Where: $\mathbf{X}^i$ represent all two-body interactions $\{S_j^i S_k^i\}_{j,k=1}^{L}$, and i runs over the samples in the dataset.
- To complete analogy: can represent the dot product by a single index $p = \{j, k\}$, i. e. $\mathbf{X}^i \cdot \mathbf{J} = X_p^i J_p$.
- Note the regression model doesn't include the minus sign.

In the following we apply ordinary least squares (OLS), Ridge and Lasso regression to the problem, and compare.

- Fig 16 shows the R^2 (=coefficient of determination, a regression performance measure. Optimal performance: $R^2 = 1$) of the three regression models.

$$R^2 = 1 - \frac{\sum_{i=1}^{n} |y_i^{true} - y_i^{pred}|^2}{\sum_{i=1}^{n} |y_i^{true} - \frac{1}{n}\sum_{i=1}^{n} y_i^{pred}|^2} \tag{81}$$

- A few remarks:
  - The regularization parameter λ affects the Ridge and LASSo regressions at scales seperated by a few orders of magnitude. Therefore, considered good practice to always check performance for the given model and data as a func of λ.

  - While the OLS and Ridge regression test curves are monotonic, the LASSO test curve is not - suggesting an optimal LASSO regularization parameter is $\lambda \approx 10^{-2}$. At this sweet spot, the Ising interaction weights $\mathbf{J}$ contains only nearest-neighbour terms (as did the model the data was generated from).

- Choosing between Ridge and LASSO in this case = similar to fixing gauge degrees of freedom.
  - Recall the uniform nearest-neighbour interaction strength $J_{j,k} = J$ which we used to generate the data, was set to unity, $J = 1$.
  - Moreover, $J_{j,k}$ was NOT defined to be symmetric (we only used the $J_{j,j+1}$, but never the $J_{j,j-1}$ elements).
- Fig 17 shows the matrix representation of the learned weights $J_{j,k}$. OSL and Ridge learn nearly symmetric weights $J \approx -0.5$. Not surprising, since it amounts to taking into account both the $J_{j,j+1}$ and $J_{j,j-1}$ terms, and the weights are distributed symmetrically between them.
- LASSO, on the other hand, tends to break this symmetry.
- Thus, we see how different regularization schemes can lead to learning equivalent models but in different "gauges".
- Any info we have about symmetry of the unknown model that generated the data should be reflected in the definition of the model and the choice of regularization.

6.5 Convexity of a regularizer

Mentioned previously the analytical LASSO solution can be found by invoking its convexity. We here provide an intro to **convexity theory**. Recall

- A set $C \subseteq \mathbb{R}^n$ is *convex* if for any $x, y \in \mathcal{C}$ and $t \in [0, 1]$,

$$tx + (1-t)y \in C \tag{82}$$

  Aka, every line segment joining x, y lies entirely in C. A func $f : \mathbb{R}^n \to \mathbb{R}$ is called convex if its domain, $dom(f)$, is a convex set, and for any $x, y \in dom(f)$ and $t \in [0, 1]$ we have

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y) \tag{83}$$

  That is, the func lies on or below the line segment joining its evaluation oat x and y. This f is called *strictly convex* if this inequality holds strictly for $x \neq y$ and $t \in (0, 1)$.

- It turns out that *for convex functions, any local minimizer is a global minimizer.*
- Algorithmically, this means that in the optimization procedure, as long as we are "going down the hill" and agree to stop when we reach the minimum, then we have hit the global minumum.
- In addition to this, there is an abundance of rich theory regarding convex duality and optimality, which allow us to understand the solutions even before solving the problem itself.
- Let's examine the two regularizers Ridge and Lasso. They are both convex problems, but only Ridge is a strictly convex problem (assuming $\lambda > 0$).
- From convex theory this means we always have a unique sol for Ridge, but not necessarily for LASSO.
- It was recently shown that under mild conditions, the LASSO sol is indeed unique. Apart from this theoretical characterization, people have also introduced the notion of Elastic Net to retain the desirable properties of both LASSO and Ridge, which is now one of the standard tools for regression analysis and ML.

6.6 Bayesian formulation of linear regression

We formulate ls from a Bayesian point of view. We'll see that regularization in learning will emerge naturally as part of the Bayesian inference procedure.

From the linear regression setup, the data $\mathcal{D}$ used to fit the regression model is generated through $y = \mathbf{x}^T\mathbf{w} + \epsilon$. We often assume ϵ is a Gaussian noise w/men zero and variance σ^2. To connect linear regression to the Bayesian framework, we often write the model as

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mu(\boldsymbol{x}), \sigma^2(\boldsymbol{x})) \tag{84}$$

Aka, our regression model is defined by a conditional probability that depends not only on data $\boldsymbol{x}$, but on some model parameters $\boldsymbol{\theta}$. Fex, if the mean a linear func of $\boldsymbol{x}$ given by $\mu = \boldsymbol{x}^T\boldsymbol{w}$, and he variance is fixed $\sigma^2(\boldsymbol{x}) = \sigma^2$, then $\boldsymbol{\theta} = (\boldsymbol{w}, \sigma^2)$.

In statistics, many problems rely on estimation of some parameters of interest. Fex: suppose we're given the height data of 20 junior students from a regional high school, but we're interested in the average height of all high school juniors in the country. It's concievable the data we're given are not representative of the student population as a whole. $=$ necessary to devise a systematic way to perform reliable estimation. We here present the **Maximum Likelihood Estimation** (MLE), and show that least squares can be derived from this framework.

MLE is defined by

$$\hat{\boldsymbol{\theta}} = \text{argmax}_\theta \text{log} p(\mathcal{D}|\boldsymbol{\theta}) \tag{85}$$

Using the assumption that samples are i.i.d., we can write the *log-likelihood* as

$$l(\boldsymbol{\theta}) \equiv \text{log} p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{n} \text{log} p(y_i|\boldsymbol{x}_i, \boldsymbol{\theta}) \tag{86}$$

Inserting p as defined above we get

$$\begin{aligned} l(\boldsymbol{\theta}) &= -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{x}_i^T\boldsymbol{w})^2 - \frac{n}{2}\text{log}(2\pi\sigma^2) && (87) \\ &= -\frac{1}{2\sigma^2}||X\boldsymbol{w} - \boldsymbol{y}||_2^2 + \text{const} && (88) \end{aligned}$$

By comparing this to the expression for $\hat{\boldsymbol{w}}_{LS}$ it's clear that performing least squares is the same as maximizing the log-likelihood of this model.

What about adding regularization? We earlier introduced the *maximum a posteriori probability (MAP) estimate.* We now show it actually corresponds to regularized linear regression, where the choice of prior determines the regularization. Recall Baye's rule

$$p(\boldsymbol{\theta}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{89}$$

Now instead of maximizing the log-likelihood $l(\boldsymbol{\theta}) = \text{log}p(\mathcal{D}|\boldsymbol{\theta})$, let's maximize the log posterior $\text{log}p(\boldsymbol{\theta}|\mathcal{D})$. Invoking Baye's rules, the MAP estimator becomes

$$\hat{\boldsymbol{\theta}}_{MAP} \equiv \text{argmax}_\theta \text{log}p(\mathcal{D}|\boldsymbol{\theta}) + \text{log}p(\boldsymbol{\theta}) \tag{90}$$

Review of Bayes terms:

- Likelihood function = $p(\boldsymbol{X}|\boldsymbol{w})$
- Prior distribution = $p(\boldsymbol{w})$
- Posterior distribution = $p(\boldsymbol{w}|\boldsymbol{X})$

Suppose we use the Gaussian prior (= the *conjugate prior* that gives a Gaussian poserior. For a given likelihood, conjugacy guarantees the preservation of prior distribution at the posterior level. Fex: for a Gaussion(Geometric) likelihood with a Gaussian(Beta) prior, the posterior distribution is still Gaussian(Beta) dist.) with zero mean and variance τ^2, namely $p(\boldsymbol{w}) = \prod_j \mathcal{N}(w_j|0, \tau^2)$, we can recast the MAP estimator into (constant terms that don't depend on the parameters have been dropped)

$$\begin{aligned} \hat{\boldsymbol{\theta}}_{MAP} &= \text{argmax}_\theta[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \boldsymbol{x}_i^T\boldsymbol{w})^2 - \frac{1}{2\tau^2}\sum_{j=1}^{n} w_j^2] && (91) \\ &= \text{argmax}_\theta[-\frac{1}{2\sigma^2}||X\boldsymbol{w} - \boldsymbol{y}||_2^2 - \frac{1}{2\tau^2}||\boldsymbol{w}||_2^2] && (92) \end{aligned}$$

The equivalence between MAP estimation with a Gaussian prior and Ridge regression is established by comparing this expression to the one for $\hat{\boldsymbol{w}}_{Ridge}(t)$ with $\equiv \sigma^2/\tau^2$. There's an analogous derivation for LASSO.

6.7 Recap and a general perspective on regularizers

In this section we

- Explored least square regression with and without regularization
- Motivated the need for regularization due to poor generalization, in particular the "high-dimensional limit" ($p \gg n$).
- Instead of showing the in-sample and out-of-sample errors explicitly, we conducted numerical experiments in Notebook 3 on the diabetes dataset and showed that regularization typically leads to better generalization.
- Due to the equivalence between the constrained and penalized form of regularized regression (in LASSO and Ridge, but not generally true in cases such as L^0 penalization), we can regard the regularized regression problem as an un-regularized problem but on a constrained set of parameters.
- Since the size of the allowed parameter space (e. g. $\boldsymbol{w} \in \mathbb{R}^p$ when un-regularized vs. $\boldsymbol{w} \in C \subset \mathbb{R}^p$ when regularized) is roughly a proxy for model complexity, solving the regularized problem is in effect solving the un-regularized problem with a smaller model complexity class $=$ we're less likely to overfit.
- We also showed connection (useing a regularization function $\Leftrightarrow$ the use of priors in Bayesian inference). It gives us more intuition about why regularization implies we're less likely to overfit the data:
  - Say you're a physics student taking a lab class. Experiment goal is to measure the behavior of different pendula and use it that to predict the formula/model that determines the period of oscillation. You would record many things (inc the lenght and mass) to give yourself the best possible chance of determining the unknown relationship, the room temperature, air currents, table vibrations etc.
  - $=$ You have a high dimensional dataset. Even higher - you're aware of time of day, that it's Wednesday, your friends in attendance, the location, etc. You didn't write it down because of strongly held prior beliefs that none of those things affect the physics taking place in the room.
  - What's serving you here is the intuition that probably only a few things matter in the physics of pendula. You're approaching the experiment with prior beliefs about how many features you'll need to pay attention to in order to predict what'll happen when you swing an unknown pendulum.

  Point being that we live in a high-dimensional world of info and while we have have good intuition in the pendulum case/well-known problems, often in the ML field we cannot say with any confidence a priori *what* the small list of things to write down will be, but we can at least use regularization to help us enforce that the list not be too long so that we don't end up predicting that the pendulum's period depends on Bob having a cold on Wednesdays.

- In both LASSO and Ridge there is a parameter λ involved. In principle, this **hyperparameter** is usually predetermined = not part of the regression process. We saw our learning performance and solution depends strongly on λ = vital to choose it properly. As discussed, one approach is to assume an *uninformative prior* on the hyperparameters, $p(\lambda)$, and average the posterior over all choices of λ following this distribution. Hoewever, large computational cost. = simpler to choose the regularization parameter through some optimization procedure.

- Emphasize: linear regression can be applied to model non-linear relationship between input and response. Done by replacing the input $\boldsymbol{x}$ w/some nonlinear func $\phi(\boldsymbol{x})$. Doing so preserves the linearity as a func of the parameters $\boldsymbol{w}$, since model is defined by the inner product $\phi^T(\boldsymbol{x})\boldsymbol{w}$. This method = *basis function expansion.*

- Recent years: surge of interest in understadning generalized linear regression models from a statistical physics perspective. Much has focused on understanding high-dimensional linear regression and compressed sensing. On a technical level, this research imports and extends the machinery of spin glass physics (replica method, cavity method, and message passing) to analyze high-dimensional linear models. A rich area of activity at the intersection of physics, computer science, information theory and ML.

7 Logistic regression

Have so far focused on continuous output. Now: classification: outcomes=discrete variables (i.e. categories). Fex, detect whether cat/dog in a picture. Or given a spin config, say the 2D Ising model, identify it's phase (e.g. ordered/disordered). Logistic regression deals with binary, dichotomous(= dividing into two parts) outcomes (e. g. true/false, success/failure, etc). Worth noting: logistic regression also commonly used in modern supervised deep learning models. Section structure:

- Define logistic regression and derive its corresponding cost function (the cross entropy) using a Bayesian approach and discuss its minimization.

- Generalize logistic regression to the case of multiple categories which is called *Sofrmax regression.*

- Demonstrate logistic regression via application to three different problems:

  - Classifying phases of the 2D Ising model

  - The SUSY dataset
  - MNIST handwritten digit classification

In this section, we consider the case where

- The dependent variables $y_i \in \mathbb{Z}$ are discrete and only take values from $m = 0, ..., M-1$ (i.e. M classes)
- Goal: predict the output classes from the design matrix $X \in \mathbb{R}^{n \times p}$ made of n samples, each of which bears p features. Of course primary goal = identify the classes to which new unseen samples belong.

Helpful to consider a slightly simple classifier before logistic regression:

- A linear classifier that categorizes examples using a weighted linear-combination of the features and an additive offset:

$$s_i = \boldsymbol{x}_i^T \boldsymbol{w} + b_0 \equiv \mathbf{x}_i^T \mathbf{w} \tag{93}$$

  where $\mathbf{x}_i = (1, \boldsymbol{x}_i)$ and $\mathbf{w}_i = (b_0, \boldsymbol{w}_i)$.

- This func takes values on the entire real axis. With logistic regression however the labels y_i are discrete variables.
- One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 otherwise.
- = The "perceptron" in ML.

Extremely simple model, and it's favorable in many cases (e.g. noisy data) to have a "soft" classifier that outputs the probability of a given category:

- Fex: Given $\mathbf{x}_i$, the classifier outputs the probability of being in category m.
- One such func: The logistic (or **sigmoid**) function:

$$f(s) = \frac{1}{1 + e^{-s}} \tag{94}$$

  Note that $1 - f(s) = f(-s)$ will be useful shortly.

7.1 The cross-entropy as a cost function for logistic regression

- Perceptron = example of a "hard classification": each datapoint deterministically assigned to a category.

- In many cases favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. Logistic regression is the most canonical example of a soft classifier.

- In logistic regression, the prob that a data point $\boldsymbol{x}_i$ belongs to a category $y_i = \{0, 1\}$ is given by

$$P(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{1}{1 + e^{-\mathbf{x}_i^T \mathbf{w}}} \tag{95}$$
$$P(y_i = 0|\boldsymbol{x}_i, \boldsymbol{\theta}) = 1 - P(y_i = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) \tag{96}$$

  where $\boldsymbol{\theta} = \mathbf{w}$ are the weights we wish to learn from the data.

- For intuition:
  - Consider a collection of non-interacting two-state systems coupled to a thermal bath (e.g. a collection of atoms that can be in two states).
  - Denote the state of system i by a binary variable: $y_i \in \{0, 1\}$.
  - From elementary statistical mechanics, we know that if the two states have energies ϵ_0 and ϵ_1 the prob for finding the system in a state y_i is just:

$$P(y_i = 0) = \frac{e^{-\beta\epsilon_0}}{e^{-\beta\epsilon_0} + e^{-\beta\epsilon_1}} = \frac{1}{1 + e^{-\beta\Delta\epsilon}} \tag{97}$$
$$P(y_i = 1) = 1 - P(y_i = 0) \tag{98}$$

  - Notice in these expressions, as is often the case in physics, only energy differences are observable.
  - If the difference in energies between two states is given by $\Delta\epsilon = \mathbf{x}_i^T \mathbf{w}$, we recover the expressions for logistic regression.

- We'll use this mapping between partition functions and classification to generalize the logistic regressor to soft-max regression later this section.

- Notice in terms of the logistic func, we can write

$$P(y_i = 1) = f(\mathbf{x}_i^T \mathbf{w}) = 1 - P(y_i = 0) \tag{99}$$

- We now define the cost func for logistic regression usng Maximum Likelihood Estimation (MLE). In MLE we choose parameters to maximize the prob of seeing the observed data.
  - Consider a dataset $\mathcal{D} = \{(y_i, \boldsymbol{x}_i)\}$ w/binary labels $y_i \in \{0, 1\}$ where the data points are drawn independently.

- The likelihood of seeing the data under our model is

$$P(\mathcal{D}|\mathbf{w}) = \prod_{i=1}^{n}[f(\mathbf{x}_i^T\mathbf{w})]^{y_i}[1 - f(\mathbf{x}_i^T\mathbf{w})]^{1-y_i} \tag{100}$$

  from which we compute the log-likelihood

$$\begin{aligned} l(\mathbf{w}) &= \log P(\mathcal{D}|\mathbf{w}) && (101) \\ &= \sum_{i=1}^{n} y_i \log f(\mathbf{x}_i^T\mathbf{w}) + (1 - y_i)\log[1 - f(\mathbf{x}_i^T\mathbf{w})] && (102) \end{aligned}$$

- The maximum likelihood estimator = the set of parameters that maximize the log-likelihood:

$$\hat{\mathbf{w}} = \text{argmax}_\theta \sum_{i=1}^{n} y_i \log f(\mathbf{x}_i^T\mathbf{w}) + (1 - y_i)\log[1 - f(\mathbf{x}_i^T\mathbf{w})] \tag{103}$$

- Since the cost (error) func = the negative log-likelihood, we have

$$\begin{aligned} \mathcal{C} &= -\,l(\mathbf{w}) && (104) \\ &= \sum_{i=1}^{n} -y_i \log f(\mathbf{x}_i^T\mathbf{w}) - (1 - y_i)\log[1 - f(\mathbf{x}_i^T\mathbf{w})] && (105) \end{aligned}$$

  This is known in statistics as the *cross-entropy.*

- Final note: Just as in linear regression we usually supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization.

7.2 Minimizing the cross entropy

Cross entropy = a convex function of the weights $\mathbf{w}$ = any local minimizer is a global minimizer. Minimizing the cost func leads to the equation

$$\mathbf{0} = \boldsymbol{\nabla}\mathcal{C}(\mathbf{w}) = \sum_{i=1}^{n}[f(\mathbf{x}_i^T\mathbf{w}) - y_i]\mathbf{x}_i \tag{106}$$

where used the logistic func identity $\partial_z f(z) = f(z)[1 - f(z)]$. This equation defines a transcendental equation for $\mathbf{w}$, the solution of which, unlike linear regression, cannot be written on a closed form. $\rightarrow$ Must use numerical methods such as introduced in the GD chapter to solve this optimization problem.

7.3 Examples of binary classification

7.3.1 Identifying the phases of the 2D Ising model

- Goal: show how to employ logistic regression to classify the states of the 2D Ising model according to their phase of matter.
- Hamiltonian for the classical Ising model:

$$H = -J \sum_{\langle ij \rangle} S_i S_j, \quad S_j \in \{\pm 1\} \tag{107}$$

  where the lattice site indices i, j run over all nearest neighbors of a 2D square lattice, and J is an interaction energy scale. Adopt periodic boundary conditions.
- Onsager proved this model undergoes a phase transition in the thermodynamic limit from an ordered ferromagnet with all spins alligned to a disordered phase at the critical temperature $T_c/J = 2/\log(1+\sqrt{2}) \approx 2.26$. For any finite system size, this critical point is expanded to a critical region around T_c.
- Can one train a statistical classifier to distinguish between the two phases of the Ising model? This could be used to locate the position of the critical point in more complicated models.
- Given an Ising state, we would like to classify whether it belongs to the ordered or disordered phase, without other info than the spin config itself. This categorical ML problem is well suited for logistic regression and will thus consist of recognizing whether a given state is ordered by looking at its bit configs.
- For the purposes of applying logistic regression, the 2D spin state will be flattened out to a 1D array, so won't be possible to learn info on the structure of the contiguous (=sharing a common border/touching) ordered 2D domains. Such info can be incorporated using deep CNNs, see later chapter.
- We consider a 2D Ising model on a 40×40 square lattice, and use Monte-Carlo (MC) sampling to prepare 10^4 states at every fixed temperature T out of a pre-defined set. We also assign a label to each state according to its phase: 0=disordered, 1=ordered.
- Near T_c the ferromagnetic correlation length diverges, leading to, amongst other things, critical slowing of the MC algo. Identifying phases could also be harder in this region perhaps. With this in mind, consider the following three types of states:

  - Ordered ($T/J < 2.0$)
  - Near-critical ($2.0 \leq T/J \leq 2.5$)
  - Disordered ($T/J > 2.5$)

  Use ordered and disordered states to train, then evaluate the performance on unseen ordered, disordered and near-critical states.

- We deploy the *liblinear* routine (= default for Scikit's logistic regression) and stochastic gradient descent (SGD) to optimize the logistic regression cost func with L_2 regularization.

- We define accuracy of the classifier = the percentage of correctly classified data points. Comparing the accuracy on the training and test data we can study the degree of overfitting.

- See Fig 21: Notice the small degree of overfitting as suggested by the training and testing accuracy curves being close together.

- The liblinear minimizer outperforms SGD on the training and test data, but not on the near-critical data for certain values of the regularization strength λ.

- Similar to the linear regression examples, we find there exists a sweet spot for the SGD regularization strength λ that results in optimal performance of the logistic regressor, at about $\lambda \approx 10^{-1}$

- Does the difficulty of the phase recognition depend on the temp of the queried sample? For states in the near-critical temp region (see fig 20) it's no longer easy for the human eye to distinguish between ordered and disordered $\rightarrow$ interesting to compare the training and test accuracies to the accuracy of the near-critical state predictions (recall: model has not been trained on near-critical states). Indeed the liblinear accuracy is about 7% smaller for the critical states compared to the test data.

- Important to note all of Scikit's logistic regreesion solvers have in-built regularizers. Crucial in order to prevent overfitting.

7.3.2 SUSY

- In high energy physics experiments we hope to discover new particles. Need to sift through events and classify as either a signal of a new process or particle, or as a background event from already understood Standard Model processes. We don't know for sure what process has occured - only know the final state particles. But, we can try to determine parts of phase space that will have a high percentage of signal events.

- Typically done by using a series of simple requirements on the kinematic quantities of the final state particles, fex having one or more leptons w/large momentums that are transverse to the beam line (p_T).
- Instead, here we'll use logistic regression in an attempt to find the relative prob that an event is from a signal or background event. Rather than using the kinematic quantities of final state particles directly, we'll use the output of our logistic regression to define a part of phase space that is enriched in signal events.
- The dataset has been produced using Monte Carlo simulations to contain events with two leptons (electrons or muons). Each event has the value of 18 kinematic variables ("features"): first 8 = direct measurements of final state particles (in this case the p_T, pseudo-rapidity η, and azimuthal angle ϕ of two leptons in the event and the amount of missing transverse momentum (MET) together with its azimuthal angle), last 10 = functions of the first 8 - high level features derived by physicists to help discriminate between the two classes. These high level features can be viewed as the physicist's attempt to use non-linear functions to classify the events, having been developed with formidable theoretical effort. We'll later revisit this problem with the tools of Deep Learning.
- Since we don't know the true underlying process, so our goal in these types of analysis is to find regions enriched in signal events. If we find an excess of events above what is expected, we can have confidence they come from the signal type we're searching for. $\rightarrow$ The two metrics of import are
  - The efficiency of signal selection
  - The background rejection achieved
- Often rather than thinking about just a single working point, performance is characterized by Reciever Operator Curves (or ROC curves). They plot signal efficiency (true-positives) vs. background rejection (true-negatives) as a func of some continuous variable such as a threshold. Here that variable will be the output signal probability of our logistic regression.
- Fig 22 shows examples of these outputs (Using L^2 regularization w/ a regularization parameter of 10^{-5}): x-axis=logsitic regression model's output "probability of this being a true signal event", y-axis=frequency. Two figures: one where the actual event inputs are signal events (meaning the p=1 point on the x-axis counts true-positive cases, while the p=0 point counts false-negative cases (Type II error)) and one plot where the event inputs are background and not signal (meaning the

p=1 point on the x-axis counts false-positive cases (Type I error), while the p=0 point counts true-negative cases). $\rightarrow$ Some signals events even look background like (false-false), and some background events look signal like (false-true) = further reason to characterize performance in terms of ROC curves.

- Fig 23: Examples of ROC curves using L^2 regularization for many different regularization parameters using either TensorFlow (top) or SciKit learn (bottom) when using the full set of 18 input variables. Notice
  - Minimal overfitting, partly because such a large training dataset (4.5 million events). How do they see this from the ROC curves..??
  - More importantly, the underlying data we're working with: each input variable is an important feature.
- Is there utility to this increased sophistication?
  - Recall, even to the learning algo, some signal events and background events look similar. Can illustrate this by a plot comparing the p_T spectrum of the leading and subleading leptons for both signal and background events. Fig 24 shows these two distributions. While *some* signal events are easily distinguished, many live in the same part of phase space as the background. This effect can also be seen in Fig 22.
  - How much discrimination power is obtained by simply putting different requirements on the input variables rather than using ML techniques? In order to compare this (called cut-based strategy in the HEP field) to regression, different ROC curves have been made for
    - Logistic regression with just the simple kinematic variables,
    - Logistic regression w/the full set of variables, and
    - Just putting requirements on the leading lepton p_T
  - Fig 25 shows a clear performance benefit from using logistic regression.
  - Note in the cut-based approach we have only used one variable where we could have put requirements on all of them. While putting more requirements would increase background rejection, it would also decrease signal efficiency. $\rightarrow$ The cut-based approach will never yield as strong discrimination as logistic regression.
  - Also interesting in fig 25 - the higher-order variables noticably help the ML techniques.

7.4 Softmax regression

- We generalize here from binary to multi-class classification.
- One approach is: treat the label as a vector $y_i \in \mathbb{Z}_2^M$ = a binary bit string of length M. Fex $\boldsymbol{y}_i = (1, 0, ..., 0)$ means the sample $\boldsymbol{x}_i$ belongs to class 1.
- The prob of $\boldsymbol{x}_i$ being in class m' is given by

$$P(y_{im'} = 1|\boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{e^{-\mathbf{x}_i^T \mathbf{w}_{m'}}}{\sum_{m=0}^{M-1} e^{-\mathbf{x}_i^T \mathbf{w}_m}} \tag{108}$$

  where $y_{im'} \equiv [\boldsymbol{y}_i]_{m'}$ refers to the m'-th component of vector $\boldsymbol{y}_i$. This is the **softmax** function.

- Therefore, the likelihood of this M-class classifier is simply

$$\begin{aligned} P(\mathcal{D}|\{\boldsymbol{w}_k\}_{k=0}^{M-1}) = &\prod_{i=1}^{n}\prod_{m=0}^{M-1} [P(y_{im} = 1|\boldsymbol{x}_i, \mathbf{w}_m)]^{y_{im}} \\ &\times [1 - P(y_{im} = 1|\boldsymbol{x}_i, \mathbf{w}_m)]^{1-y_{im}} \end{aligned} \tag{109}$$

- From this we similarly define the cost func:

$$\begin{aligned} E(\mathbf{w}) = -&\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im} \log P(y_{im} = 1|\boldsymbol{x}_i, \mathbf{w}_m) \\ &+ (1 - y_{im}) \log(1 - P(y_{im} = 1|\boldsymbol{x}_i, \mathbf{w}_m)) \end{aligned} \tag{110}$$

  As expected, for $M = 1$ we recover the cross entropy for logistic regression.

7.5 An example of SoftMax classification: MNIST digit classification

- The MNIST dataset: 70000 handwritten digits, each of which is laid out on a 28×28-pixel grid.
- Every pixel assumes one of 256 grayscale values, interpolating between white and black.
- 10 categories for the digits 0 through 9 $\rightarrow$ SoftMax regression with $M = 10$
- Fig 26 shows the learned weights $\boldsymbol{w}_k$ where k corresponds to class labels (i.e. digits).
- We'll come back to SoftMax in chapter 9.

8 Combining models

- One of most powerful and widely applied ideas in modern ML: ensamble methods = combine predictions from multiple, often weak, statistical models. They undergird many of the winning entries in data science competitions such as Kaggle, especially on structured datasets (neural networks generally perform better than enasmble methods on unstructured data, images, and audio).
- Even in neural nets context, it's common to combine predictions from multiple nerual nets to increase performance on tough image classification tasks.
- On one hand, idea of training multiple models, then using a weighted sum of their predictions, very natural. On the other hand, can imagine the ensamble predictions can be much worse than that of the individual models - especially when pooling reinforces weak but correlated deficiencies in each individual predictor. $\rightarrow$ Important to understand when we expect ensamble methods to work.
- To that end, will revisit the bias-variance tradeoff, generalize it to an ensamble of classifiers. Will show key to determine when ensamble methods work = the degree of correlation between the models in the ensamble.

8.1 Revisiting the bias-variance tradeoff for ensembles

- This tradeoff summarizes the fundamental tension in ML between (model complexity $\Leftrightarrow$ amount of training data needed to fit it).
- Key property emerging from this analysis: the correlation between models that constitue the ensamble. Important for two distinct reasons:
  - Holding the ensemble size fixed, averaging the predicitons of correlated models reduces the variance less than averaging uncorrelated models.
  - In some cases, correlations between models within an ensemble can result in an *increase* in bias, offsetting any potential reduction in variance gained from ensemble averaging. We'll discuss this in the context of bagging below. One of the most dramatic examples of increased bias from correlations = the catastrophic predictive failure of almost all derivative models used by Wall Street during 2008 financial crisis.

8.1.1 Bias-variance decomposition for ensembles

- We'll discuss bias-variance tradeoff in context of continuous predictions such as regression, but many ideas carry over to classification tasks.
- Review of the bias-variance tradeoff in context of a single model:
  - $\boldsymbol{X}_\mathcal{L} = \{(y_j, \boldsymbol{x}_j), j = 1...N\}$ = data in dataset
  - $y = f(\boldsymbol{x}) + \epsilon$ = noisy model from which we assume the true data is generated from. ϵ = normally distributed w/mean zero and standard deviation σ_ϵ.
  - $\hat{g}_\mathcal{L}(\boldsymbol{x})$ = a predictor formed by a statistical procedure we assume we have, e.g. least squares regression. The predictor gives our model's prediction using a dataset $\mathcal{L}$.
  - This estimator is chosen by minimizing a cost func which, for the sake of correctness, we take to be the squared error

    $$\mathcal{C}(\boldsymbol{X}, g(\boldsymbol{x})) = \sum_i (\boldsymbol{y}_i - \hat{g}_\mathcal{L}(\boldsymbol{x}_i))^2 \tag{111}$$

  - $\mathcal{L}$ = dataset drawn from some underlying distribution that describes the data.
  - $\{\mathcal{L}_j\}$ = many different datasets of the same size as $\mathcal{L}$ drawn from this distribution.
  - The corresponding $\hat{g}_{\mathcal{L}_j}(\boldsymbol{x})$ will differ from each other due to stochastic effects arising from the sampling noise. $\rightarrow$ Can view our estimator $\hat{g}_\mathcal{L}(\boldsymbol{x})$ as a random variable (technically functional) and define an expectation value $E_\mathcal{L}$ the usual way.
  - $E_\mathcal{L}$ = expectation value computed by drawing infinitely many different datasets $\{\mathcal{L}_j\}$ of the same size, fitting the corresponding estimator, then averaging over the results.
  - E_ϵ = the expectation value over different instances of the "noise" ϵ.
  - Can then decompose the expected generalization error as

    $$E_{\mathcal{L},\epsilon}[\mathcal{C}(\boldsymbol{X}, g(\boldsymbol{x}))] = Bias^2 + Var + Noise \tag{112}$$

    where the bias=the deviation of the expectation value of our estimator (=the asymptotic value of our estimator in the limit of infinite data) from the true value:

    $$Bias^2 = \sum_i (f(\boldsymbol{x}_i) - E_\mathcal{L}[\hat{g}_\mathcal{L}(\boldsymbol{x}_i)])^2 \tag{113}$$

The variance = how much our estimator fluctuates due to finite-sample effects:

$$Var = \sum_i E_{\mathcal{L}}[(\hat{g}_{\mathcal{L}}(\boldsymbol{x}) - E_{\mathcal{L}}[\hat{g}_{\mathcal{L}}(\boldsymbol{x})])^2] \tag{114}$$

The noise term = part of the error due to intrinsic noise in the data generation process that no statistical estimator can overcome:

$$Noise = \sum \sigma^2_{\epsilon_i} \tag{115}$$

Let's generalize this to ensembles of estimators:

- $\boldsymbol{X}_{\mathcal{L}}$ = a given dataset
- θ = given hyper-parameters that parametrize member of our ensemble
- We'll consider a procedure that deterministically generates a model $\hat{g}_{\mathcal{L}}(\boldsymbol{x}_i, \theta)$ given $\boldsymbol{X}_{\mathcal{L}}$ and θ.
- Assume θ includes some random parameters that introduce stochasticity into our ensemble (e.g. an initial condition for stochastic gradient descent or a random subset of features or data points used for training)
- We'll be concerned with the expected prediction error of the *aggregate ensemble predictor*

$$\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i, \{\theta\}) = \frac{1}{M}\sum_{m=1}^{M} \hat{g}_{\mathcal{L}}(\boldsymbol{x}_i, \theta_m) \tag{116}$$

- For future reference we'll define the mean, variance, covariance (= the connected correlation function in the language of physics) and the normalized correlation coefficient wrt θ of the estimators in our ensemble as

$$\begin{aligned} E_\theta[\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta)] &= \mu_{\mathcal{L},\theta}(\boldsymbol{x}) \\ E_\theta[\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta)^2] - E_\theta[\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta)]^2 &= \sigma^2_{\mathcal{L},\theta}(\boldsymbol{x}) \\ E_\theta[\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta_m)\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta_{m'})] - E_\theta[\hat{g}_{\mathcal{L}}(\boldsymbol{x}, \theta_m)]^2 &= C_{\mathcal{L},\theta_m,\theta_{m'}}(\boldsymbol{x}) \\ \rho(\boldsymbol{x}) &= \frac{C_{\mathcal{L},\theta_m,\theta_{m'}}(\boldsymbol{x})}{\sigma^2_{\mathcal{L},\theta}} \end{aligned} \tag{117}$$

By definition we assume $m \neq m'$ in $C_{\mathcal{L},\theta_m,\theta_{m'}}$.

We can now ask about the expected generalization (out-of-sample) error for the ensemble

$$E_{\mathcal{L},\epsilon,\theta}[\mathcal{C}(\boldsymbol{X}, \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}))] = E_{\mathcal{L},\epsilon,\theta}[\sum_i (\boldsymbol{y}_i - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i, \{\theta\}))^2] \tag{118}$$

As in the single estimator case, we decompose the error into a noise, bias and variance term. To see this, note that

$$\begin{aligned}
E_{\mathcal{L},\epsilon,\theta}[\mathcal{C}(\boldsymbol{X},\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}))] =& E_{\mathcal{L},\epsilon,\theta}[\sum_i(\boldsymbol{y}_i - f(\boldsymbol{x}_i) + f(\boldsymbol{x}_i) - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2] \quad (119)\\
=& \sum_i E_{\mathcal{L},\epsilon,\theta}[(\boldsymbol{y}_i - f(\boldsymbol{x}_i))^2 + (f(\boldsymbol{x}_i) - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2\\
&+ 2(\boldsymbol{y}_i - f(\boldsymbol{x}_i))(f(\boldsymbol{x}_i) - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))] \quad (120)\\
=& \sum_i \sigma^2_{\epsilon_i} + \sum_i E_{\mathcal{L},\theta}[(f(\boldsymbol{x}_i) - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2] \quad (121)
\end{aligned}$$

where in the last line we used that $E_\epsilon[y_i] = f(\boldsymbol{x}_i)$ (why is only θ in the subindex of the E here?? How do the arithmetics of expec. values over several variables work? well probably the logical way, i.e. $E_{\mathcal{L},\epsilon,\theta} = (E_{\mathcal{L}} + E_\epsilon + E_\theta)/3$ and probably ϵ was the only one they bothered to include here b.c. ϵ is the only variable we would expect $y = f(\boldsymbol{x}) + \epsilon$ to change over/be affected by) to eliminate the last term. Further decompose the second term as

$$\begin{aligned}
E_{\mathcal{L},\theta}[(f(\boldsymbol{x}_i) - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2] =& E_{\mathcal{L},\theta}[(f(\boldsymbol{x}_i) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})]\\
&+ E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})] - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2] \quad (122)\\
=& E_{\mathcal{L},\theta}[(f(\boldsymbol{x}_i) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})])^2]\\
&+ E_{\mathcal{L},\theta}[(E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})] - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))^2]\\
&+ 2E_{\mathcal{L},\theta}[(f(\boldsymbol{x}_i) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})])\\
&\quad (E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})] - \hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}))] \quad (123)\\
=& (f(\boldsymbol{x}_i) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})])^2\\
&+ E_{\mathcal{L},\theta}[(\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\}) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}_i,\{\theta\})])^2] \quad (124)\\
=& Bias^2(\boldsymbol{x}_i) + Var(\boldsymbol{x}_i) \quad (125)
\end{aligned}$$

where we defined the bias and variance of the aggregate predictor as

$$\begin{aligned}
Bias^2(\boldsymbol{x}) \equiv& (f(\boldsymbol{x}) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\})])^2 \quad (126)\\
Var(\boldsymbol{x}) \equiv& E_{\mathcal{L},\theta}[(\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\}) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\})])^2] \quad (127)
\end{aligned}$$

So far, ensemble calc almost identical to that of a single estimator. But, since the aggregate estimator is a sum of estimators, its variance implicitly depends on the correlations between the individual estimators in the ensemble. Using the definition of the aggregate estimator $\hat{g}^A_{\mathcal{L}}$ and those of the mean, variance, covariance and normalized correlation coefficients given earlier, we get

$$
\begin{aligned}
Var(\boldsymbol{x}) =& E_{\mathcal{L},\theta}[(\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\}) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\})])^2] && (128)\\
=& \frac{1}{M}[\sum_{m,m'} E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\theta_m)\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\theta_{m'})] - M^2\sum_i [\mu_{\mathcal{L},\theta}(\boldsymbol{x})]^2] && (129)\\
=& \rho(\boldsymbol{x})\sigma^2_{\mathcal{L},\theta} + \frac{1-\rho(\boldsymbol{x})}{M}\sigma^2_{\mathcal{L},\theta} && (130)
\end{aligned}
$$

This formula = key to udnerstand the power of random ensembles. By using large ensembles ($M \to \infty$), can significantly reduce the variance, and for completely random ensembles where the models are uncorrolated ($\rho(\boldsymbol{x}) = 0$), maximally surpress the variance! $\to$ Using the aggregate predictor beats down fluctuations due to finite-sample effects. Key = decorrelate the models as much as possible while still using a very large ensemble.

Can be worried this comes at the expense of a large bias. But no. When models in the ensemble completely random, the bias of the aggregate predictor = the expected bias of a single model:

$$
\begin{aligned}
Bias^2(\boldsymbol{x}) =& (f(\boldsymbol{x}) - E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\{\theta\})])^2 && (131)\\
=& (f(\boldsymbol{x}) - \frac{1}{M}\sum_{m=1}^{M} E_{\mathcal{L},\theta}[\hat{g}^A_{\mathcal{L}}(\boldsymbol{x},\theta_m)])^2 && (132)\\
=& (f(\boldsymbol{x}) - \mu_{\mathcal{L},\theta})^2 && (133)
\end{aligned}
$$

$\to$ for a random ensemble one can always add more models without increasing the bias. This observation lies behind the immense power of random forest methods. For other methods inc bagging, we'll see that the bootstrapping procedure actually does increase the bias - but in many cases the increase is negligible compared to the reduction in variance.

8.1.2 Summarizing the theory and intuitions behind ensembles

Three distinct shortcomings fixed by ensemble methods: statistical, computational, representational.

- Statistical: When learning set too small, learning algo can typically find several models in the hypothesis space $\mathcal{H}$ that all give the same performance on training data. Provided their predictions uncorrelated, averaging several models reduces risk of choosing wrong hypothesis.

- Computational: Many learning algos rely on some greedy assumption/ local search that may get stuck in local optima. $\to$ An ensemble made of individual models built from many different starting points may provide a better approximation of the true unknown function than any of the single models.

- Representational: In most cases, for a learning set of finite size, the true func cannot be represented by any of the candidate models in $\mathcal{H}$. By combining several models in an ensemble, may be possible to expand the space of representable funcs and better model the true func.

Combining models may come at the price of introducing mroe parameters to our learning procedure. But if the problem itself can never be learned through a simple hypothesis, then no reason to avoid applying a more complex model. Ensemble methods reduce the variance and often easier to train than a single complex model = powerful way of increasing representational power/expressivity.

Our analysis gives several intuitions for how we should construct ensembles.

- Should try to randomize ensemble construction as much as possible to reduce correlations between predictors in the ensemble. Ensures our variance reduced while minimizing increase in bias due to correlated errors.

- The ensembles will work best for procedures where the error of the predictor is dominated by the variance and not the bias. $\rightarrow$ These methods are especially well suited for unstable procedures whose results sensitive to small changes in the training data.

The ideas that using an ensemble allows us to reduce variance and that the procedure works best for unstable predictors which errors are dominated by variance caused by finite sampling rather than bias, may be carried over to classification tasks, even if this section has focused on continuous predictors such as regression.

8.2 Bagging

BAGGing = Bootstrap AGGregation = one of the simplest and most used ensemble methods. Imagine we have

- $\mathcal{L}$ = a very large dataset that we could partition into M smaller datasets which we label $\{\mathcal{L}_1, ..., \mathcal{L}_M\}$. If each partition is sufficiently large to learn a predictor, we can create an ensemble aggregate predictor, composed of predictors trained on each subset of the data.

- For continuous predictors like regression, this is just the average of all the individual predictors:

$$\hat{g}^A_{\mathcal{L}} = \frac{1}{M} \sum_{i=1}^{M} g_{\mathcal{L}_i}(\boldsymbol{x}) \tag{134}$$

- For classification tasks where each predictor predicts a class label $j \in \{1, ..., J\}$, this is just the majority vote of all the predictors,

$$\hat{g}^A_{\mathcal{L}}(\boldsymbol{x}) = \text{argmax}_j \sum_{i=1}^{M} I[g_{\mathcal{L}_i}(\boldsymbol{x}) = j] \tag{135}$$

  where $I[g_{\mathcal{L}_i}(\boldsymbol{x}) = j]$ is an indicator func that is equal to one if $g_{\mathcal{L}_i}(\boldsymbol{x}) = j$ and zero otherwise. From our theoretical discussion we know this may significantly reduce the variance without increasing the bias.

- This form of aggregation clearly only works if enough data in each partition set $\mathcal{L}_i$. Consider the extreme limit where $\mathcal{L}_i$ contains exactly one point. In this case, the base hypothesis $g_{\mathcal{L}_i}(\boldsymbol{x})$ (e.g. linear regressor) becomes extremely poor and the procedure above fails. One way to circumvent this shortcoming: resort to **empirical bootstrapping**, a resampling technique in statistics. Idea is to use sampling w/replacement to create new "bootstrapped" datasets $\{\mathcal{L}_1^{BS}, ..., \mathcal{L}_M^{BS}\}$ from our original datasets $\mathcal{L}$. These bootstrapped datasets share many points, but due to the sampling w/replacement, are all somewhat different from each other.

- In the bagging procedure, we create an aggregate estimator by replacing the M independent datasets by the M bootstrapped estimators:

$$\hat{g}^{BS}_{\mathcal{L}}(\boldsymbol{x}) = \frac{1}{M} \sum_{i=1}^{M} g_{\mathcal{L}_i^{BS}}(\boldsymbol{x}) \tag{136}$$

  and

$$\hat{g}^{BS}_{\mathcal{L}}(\boldsymbol{x}) = \text{argmax}_j \sum_{i=1}^{M} I[g_{\mathcal{L}_i^{BS}}(\boldsymbol{x}) = j] \tag{137}$$

- This bootstrapping procedure allows us to construct an approximate ensemble and thus reduce variance. For unstable predictors, this can significantly improve performance.

- Price we pay for using bootstrapped training datasets is an increase in the bias of our bagged estimators: note that as the number of datasets $M \to \infty$, the expectation wrt the bootstrapped samples converges to the empirical distribution describing the training data set $p_{\mathcal{L}}(\boldsymbol{x})$ (e.g. a delta func at each datapoint in $\mathcal{L}$) = in general different from the true generative distribution for the data $p(\boldsymbol{x})$.

- Fig 29: Bagging with a perceptron (linear classifier) as the base classifier that constitutes the elements of the ensemble. See that although each individual classifier in the ensemble performs poorly, bagging these estimators yields reasonably good performance. Raises questions like *why bagging works* and *how many bootstrap samples are needed.* As mentioned,

- Bagging is effective on "unstable" learning algos = small changes in the training set result in large changes in predictions.
- When the procedure is unstable, the prediction error is dominated by the variance and one can exploit the aggregation component of bagging to reduce the prediction error.
- In contrast, for a stable procedure the accuracy is limited by the bias introduced by using bootstrapped datasets.
- = There is an instability-stability transition point beyond which bagging stops improving our prediction.

Brief introduction to bootstrapping:

- $\mathcal{D} = \{X_1, ..., X_n\}$ = a finite set of n data points given as training samples
- Our job: construct measures of confidence for our sample estimates (e.g. the confidence interval, or mean-squared error of sample median estimator).
- One then first samples n points **with replacement** from $\mathcal{D}$ to get a new set $\mathcal{D}^{\star(1)} = \{X_1^{\star(1)}, ..., X_n^{\star(1)}\}$, called a **bootstrap sample**, which possibly contains repetitive elements.
- Then repeat the same procedure to get in total B such sets: $\mathcal{D}^{\star(1)}, ..., \mathcal{D}^{\star(B)}$.
- Next step: use these B bootstrap sets to get the **bootstrap estimate** of the quantity of interest.
- Fex: let $M_n^{\star(k)} = Median(\mathcal{D}^{\star(k)})$ be the sample median of bootstrap data $\mathcal{D}^{\star(k)}$.
- Then we can construct the variance of the distribution of bootstrap medians as:

$$\hat{Var}_B(M_n) = \frac{1}{B-1}\sum_{k=1}^{B}(M_n^{\star(k)} - \bar{M}_n^{\star})^2 \tag{138}$$

where

$$\bar{M}_n^{\star} = \frac{1}{B}\sum_{k=1}^{B} M_n^{\star(k)} \tag{139}$$

is the mean of the median of all bootstrap samples.

- Specifically it has been shown that in the $n \to \infty$ limit, the distribution of the bootstrap estimate will be a Gaussian centered around $\hat{M}_n(\mathcal{D}) = Median(X_1, ..., X_n)$ w/standard deviation $\propto 1\sqrt{n}$.
- $\to$ The bootstrap distribution $\hat{M}_n^\star - \hat{M}_n$ approximates fairly well the sampling distribution $\hat{M}_n - M$ from which we obtain the training data $\mathcal{D}$.
- Note: M = the median based on which the true dist $\mathcal{D}$ is generated. Aka, if we plot the histogram of $\{M_n^{\star(k)}\}_{k=1}^B$, we'll see that in the large n limit it can be well fitted by a Gaussian which sharp peaks at $\hat{M}_n(\mathcal{D})$ and vanishing variance defined by $\hat{Var}_B$ above.
- Fig 28: illustration.
- An interpretation of all the Ms involved here:
  - M = the median based on which the true distribution/sample $\mathcal{D}$ is generated
  - M_n = ?
  - $\hat{M}_n = \hat{M}_n(\mathcal{D}) = Median(X_1, ..., X_n)$ = the median of the sample $\mathcal{D}$
  - $M_n^{\star(k)} = Median(\mathcal{D}^{\star(k)})$ = the sample median of bootstrap data $\mathcal{D}^{\star(k)}$.
  - $\bar{M}_n^\star = \frac{1}{B}\sum_{k=1}^B M_n^{\star(k)}$ = the mean of the median of all bootstrap samples.
  - $\hat{M}_n^\star$ = ?
  - $\hat{M}_n - M$ = the sampling distribution from which we obtain the training data $\mathcal{D}$ = ?
  - $\hat{M}_n^\star - \hat{M}_n$ = bootstrap distribution = $\approx \hat{M}_n - M$ = ?

8.3 Boosting

- In bagging, the contribution of all predictors weighted equally in the bagged (aggregate) predictor. But in principle a myriad ways to combine different predictors. Sometimes might prefer autocratic approach that emphasizes the best predictors, other times may be better to opt for more "democratic" ways as is done in bagging.
- In boosting en ensemble of weak classifiers $\{g_k(\boldsymbol{x})\}$ is combined into an aggregate, boosted classifier. But unlike bagging, each classifier is associated with a weight α_k indicating how much it contributes to the

aggregate classifier

$$g_A(\boldsymbol{x}) = \sum_{K=1}^{M} \alpha_k g_k(\boldsymbol{x}) \tag{140}$$

where $\sum_k \alpha_k = 1$.

- For already discussed reasons, boosting, like all ensemble methods, works best when combining simple, high-variance classifiers into a more complex whole.
- Here focus on "adaptive boosting" or AdaBoost. Basic idea: form the aggregate classifier in an iterative process.
- Important: at each iteration we reweight the error function to reinforce data points where the aggregate classifier performs poorly. In this way we can successively ensure our classifier has good performance over the whole dataset.
- Discussion of AdaBoost procedure in greater detail:
  - $\mathcal{L} = \{(\boldsymbol{x}_i, y_i), i = 1, ..., N\}$ a given dataset where $\boldsymbol{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y} = \{+1, -1\}$
  - Our objective: find an optimal hypothesis/classifier $g : \mathcal{X} \to \mathcal{Y}$ to classify the data.
  - Let $\mathcal{H} = \{g : \mathcal{X} \to \mathcal{Y}\}$ be the family of classifiers available in our ensemble.
  - In the AdaBoost we're concerned with the classifiers that perform somehow better than "tossing a fair coin". $=$ For each classifier, the family $\mathcal{H}$ can predict y_i correctly at least half the time.
  - Construct the boosted classifier as follows:
    * **Initialize** $w_{t=1}(\boldsymbol{x}_n) = 1/N, n = 1, ..., N$
    * **For** $t = 1, ..., T$, **do**:
      · Select from $\mathcal{H}$ a hypothesis g_t that minimizes the weighted error

      $$\epsilon_t = \sum_{i=1:g_t(\boldsymbol{x}_i)\neq y_i}^{N} w_t(\boldsymbol{x}_i) \tag{141}$$

      · Let $\alpha_t = \frac{1}{2}\ln\frac{1-\epsilon_t}{\epsilon_t}$, update the weight for each data $\boldsymbol{x}_n$ by

      $$w_{t+1}(\boldsymbol{x}_n) \leftarrow w_t(\boldsymbol{x}_n)\frac{e^{-\alpha_t y_n g_t(\boldsymbol{x}_n)}}{Z_t} \tag{142}$$

      where $Z_t = \sum_{n=1}^{N} w_t(\boldsymbol{x}_n)e^{-\alpha_t y_n g_t(\boldsymbol{x}_n)}$ ensures all weights add up to unity.

      · **Output** $g_A(\boldsymbol{x}) = sign(\sum_{t=1}^{T} \alpha_t g_t(\boldsymbol{x}))$

8.4 Random forests

- A random forest is composed of a family of (randomized) tree-based classifier decision trees (= high-variance, weak classifiers that can be easily randomized, and as such, are ideally suited for ensemble-based methods).
- A **decision tree** uses a series of questions to hierarchically partition the data. Each branch consists of a question that splits the data into smaller subsets (e.g. is some feature larger than a given number?), with leaves (end points) of the tree corresponding to the ultimate partitions of the data.
- Goal: construct trees such that the partitions are informative about the class label (see fig 30). More complex decision trees lead to finer partitions that give improved performance on the training set. But, this generally leads to overfitting, limiting the out-of-sample performance. $\rightarrow$ Almost all decision trees use some form of regularization (e.g. maximum depth of the tree) to control complexity and reduce overfitting.
- Decision trees have extremely high variance, and often extremely sensitive to many details of the training data. Not surprising since they're learned by partitioning the training data. $\rightarrow$ Individual trees are weak classifiers. But, these same properties make them ideal for incorporation in an ensemble method.
- To create ensemble of trees, must introduce a randomization procedure: Power of ensembles to reduce variance only manifests when randomness reduces correlations between the classifiers within the ensemble. Randomness usually introduced into random forests in one of three distinct ways.
  - First: Use bagging and simply "bag" the decision trees by training each decision tree on a different bootstrapped dataset. Strictly speaking this procedure does not constitute a random forest but rather **bagged decision trees**.
  - Second: Only use a different random subset of the features at each split in the tree. This "feature bagging" is the distinguishing characteristic of random forests. It reduces correlations between decision trees that can arise when only a few features are strongly predictive of the class label.
  - Finally: Extremized random forests (ERFs) combine ordinary and feature bagging with an extreme randomization procedure where splitting is done randomly instead of using optimality criteria.

Even though this reduces the predictive power of each individual decision tree, it still often improves the predictive power of the ensemble because it dramatically reduces correlations between members and prevents overfitting.

- Fig 31: Examples of the kind of decision surfaces found by decision trees, random forests, and Adaboost.

8.5 Gradient boosted trees and XGBoost

- Idea: Use intuition from boosting and gradient descent (in particular Newton's method) to construct ensembles of decision trees. As in boosting, the ensembles are created by iteratively adding new decision trees to the ensemble.
- In gradient boosted trees, a central role is played by a cost func measuring the performance of our ensemble. At each step, we compute the gradient of the cost func wrt the predicted value of the ensemble and add trees that move us in the dir of the gradient.
- Ofc, this requires clever way of mapping gradients to decision trees. We give a brief overview of how this is done within XGboost.
- Starting point: A clever parametrization of decision trees (dts). We here use notation where the dt make cont. predictions (regression trees), though this can easily be generalized to classification tasks.
- Parametrize a decision tree $g_j(\boldsymbol{x})$ with K leaves by two quantities:
  - $q(\boldsymbol{x})$ = a func that maps each data point to one of the leaves of the tree, $q : \boldsymbol{x} \in \mathbb{R}^d \to \{1, 2, ..., K\}$
  - $\boldsymbol{w} \in \mathbb{R}^T$ that assigns a predicted value to each leaf.

  In other words, the dt's prediction for the datapoint $\boldsymbol{x}_i$ is simply: $q(\boldsymbol{x}_i) = w_{q(\boldsymbol{x}_i)}$
- In addition to a parametrization of dts, we also must specify a cost func which measures predictions. The prediction of our ensemble for a datappoint $(y_i, \boldsymbol{x}_i)$ is given by

  $$\hat{y}_i = g_A(\boldsymbol{x}_i) = \sum_{j=1}^{M} g_j(\boldsymbol{x}_i) \tag{143}$$

  where $g_j(\boldsymbol{x}_i)$ is the prediction of the j-th dt on on datapoint $\boldsymbol{x}_i$, and M is the number of members in the ensemble.

  As discussed in the context of random trees above, without regularization, dts tend to overfit the data by dividing it into smaller and smaller partitions. Thus, our cost func is generally composed of two terms:

- A term that measures the goodness of predictions on each datapoint, $l_i(y_i, \hat{y}_i)$, which is assumed to be differentiable and convex
- And for each tree in the ensemble, a regularization term $\Omega(g_j)$ that does not depend on the data

$$\mathcal{C}(\boldsymbol{X}, g_A) = \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \sum_{j=1}^{M} \Omega(g_j) \tag{144}$$

where index i runs over data points and j runs over dts in our ensemble. In XGBoost, the regularization func is chosen to be

$$\Omega(g) = \gamma T + \frac{\lambda}{2} \sum_{l=1}^{T} w_l^2 \tag{145}$$

with γ and λ regularization parameters that must be chosen appropriately. Notice this regularization penalizes both large weights on the leaves (similar to L^2-regularization) and having large partitions w/many leaves.

- As in boosting, we form the ensemble iteratively. For this reason we define a family of predictors $\hat{y}_t$ as

$$\hat{y}_i^{(t)} = \sum_{j=1}^{t} g_j(\boldsymbol{x}_i) = \hat{y}_i^{(t-1)} + g_t(\boldsymbol{x}_i) \tag{146}$$

Note: by definition $y_i^{(M)} = g_A(\boldsymbol{x}_i)$. The central idea is that for large t, each dt is a small perturbation to the predictor (of order $1/K$) and hence we can perform a Taylor expansion on our loss func to second order:

$$\mathcal{C}_t = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + g_t(\boldsymbol{x}_t)) + \Omega(g_t) \tag{147}$$

$$\approx \mathcal{C}_{t-1} + \Delta\mathcal{C}_t \tag{148}$$

with

$$\Delta\mathcal{C}_t = b_i l(y_i, \hat{y}_i^{(t-1)}) g_t(\boldsymbol{x}_i) + \frac{1}{2} a_i g_t(\boldsymbol{x}_i)^2 + \Omega(g_t) \tag{149}$$

where

$$a_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \tag{150}$$

$$b_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \tag{151}$$

We then choose the t-th dt g_t to minimize $\Delta\mathcal{C}_t$. Almost identical to how we derived the Newton method update in the gradient descent section.

- Can actually derive an expression for the parameters of g_t that minimize $\Delta\mathcal{C}_t$ analytically. To simplify notation: define
  - The set of points $\boldsymbol{x}_i$ that get mapped to leaf $j : I_j = \{i : q_t(\boldsymbol{x}_i) = j\}$
  - The functions $B_j = \sum_{i\in I_j} b_i$ and $A_j = \sum_{i\in I_j} a_i$.

  In terms of these quantities we can write

$$\Delta\mathcal{C}_t = \sum_{j=1}^{T}[B_j w_j + \frac{1}{2}(A_j + \lambda_j)w_j^2] + \lambda T \tag{152}$$

  where we made the t-dependence of all parameters implicit.

- To find optimal w_j, just as in Newton's method take gradient of the above expression wrt w_j and set this equal to zero, to get

$$w_j^{opt} = -\frac{B_j}{A_j + \lambda} \tag{153}$$

  plugging back into $\Delta\mathcal{C}_t$ gives

$$\Delta\mathcal{C}_t^{opt} = -\frac{1}{2}\sum_{j=1}^{K}\frac{B_j^2}{A_j + \lambda} + \gamma T \tag{154}$$

  $\rightarrow$ Clear that $\Delta\mathcal{C}_t^{opt}$ measures the in-sample performance of g_t and we should find the dt that minimizes this value. Could in principle enumerate all possible trees over the data and find the tree that minimizes $\Delta\mathcal{C}_t^{opt}$.

- However, in practice, this is impossible. Instead, an approximate greedy algo is run that optimizes one level of the tree at a time by trying to find optimal splits of the data. Leads to a tree that is a good local minimum of $\Delta\mathcal{C}_t^{opt}$ which is then added to the ensemble.

- Emphasize this is only high level sketch of how the algo works. In practice, additional regularization such as shrinkage and feature subsampling is also used. In addition, there are many numerical and technical tricks used for the approximate algo and how to find splits of the data that give good dts.

8.6 Application to the Ising model and Supersymmetry Datasets

- Now illustrate with using two physics examples (we previously analyzed both using logistic regression):

  - Classifying the phases of the spin configs of the 2D-Ising model above and below the critical temp using random forests and
  - Classifying Monte-Carlo simulations of collision events in the SUSY dataset as supersymmetric or standard using an XGBoost implementation of gradient-boosted trees.

- We show that on the Ising dataset, the RFs perform significantly better than logistic regression models whereas gradient boosted trees seem to yield an accuracy of about 80%, comparable to published results.
- Ising: We assign a label to each state according to its phase: 0=disordered, 1=ordered. Divide the dataset into three categories according to the temp at which samples are drawn: ordered ($T/J < 2.0$), near-critical ($2.0 \leq T/J \leq 2.5$) and disordered ($T/J > 2.5$).
- Use the ordered and disordered states to train a random forest, evaluate our learned model on a test set of unseen ordered and disordered states. Also ask how well our RF can predict the phase of samples drawn in the region (=predict whether the temp of a critical sample is above or below the critical temp). Since our model never trained on critical region samples, prediction in this region is a test of the algo's ability to generalize to new regions in phase space.
- Result in Fig 32. Used two types of RFs: one where the ensemble consists of coarse dts w/few leaves, another w/finer dts w/many leaves. Extremely high accuracy on training and test sets (over 99%) for both.
- However, notice the RF consisting of coarse trees perform extremely poorly on samples from the critical region whereas RF w/fine trees classifies critical samples w/accuracy of nearly 85%.
- Interestingly, and unlike with logistic regression, this performance in the critical region requires almost no parameter tuning. This because, as discussed above, RFs are largely immune to overfitting problems even as the number of estimators in the ensemble becomes large. Increasing the number of estimators in the ensemble does increase performance - but at a large cost in computational time.
- SUSY: used the XGBoost implementation of gradient boosted trees to classify Monte-Carlo collisions from the SUSY dataset.
- With default parameters using a small subset of the data (100 000 out of the full 5 million samples), we achieved accuracy of 79%, which could be improved to nearly 80% after some fine tuning.
- Comparable to published results and those we obtained using logistic regression.

- One nice feature of ensemble methods such as XGBoost is that they automatically allow us to calc feature scores (Fscores) that rank the importance of various features for classification. Higher the Fscore, the more important the feature. Fig 33 shows feature scores for the production of electrically-charged supersymmetric particles ($\chi\pm$) which decay into W bosons and an electrically neutral supersymmetric particle χ^0, which is invisible to the detector.
- The features are a mix of eight directly measurable quantities, as well as ten hand crafted features chosen using physics knowledge. Consistent with the physics of these supersymemtric decays in the lepton channel, we find the most informative features for classification are the missing transverse energy along the vector defined by the charged leptons (Axial MET) and the missing energy magnitude due to χ_0.

9 An introduction to Feed-Forward Deep Neural Networks (DNNs)

- Long history, but remerged to prominence after a rebranding as "Deep Learning" in the mid 2000s.
- Truly caught attention in 2012 when Alex Krizhevsky, Ilya Sutskever and Geoff Hinton used a GPU-based DNN model (AlexNet) to lower the error rate on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by an incredible twelve percent from 28% to 16%. Three years later an ML group from Microsoft achieved an error of 3.57% using an ultra-deep residual neural network (ResNet) with 152 layers. Since then, DNNs have become the workhorse technique for many image and speech recognition based ML tasks. Given rise to a number of high-level libraries and packages (Caffe, Kera, Pytorch, TensorFlow).
- Conceptually helpful to divide neural networks into four categories:
  - General purpose neural networks for supervised learning
  - Neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs)
  - Neural networks for sequential data such as Recurrent Neural Networks (RNNs)
  - Neural networks for unsupervised learning such as Deep Boltzmann Machines.

In this chapter we'll discuss the first, and CNN and DPM in later chapters. While increasingly important for many applications, we omit RNNs.

- As with most intellectual fields experiencing rapid expansion, many commonly accepted heuristics turn out not to be as powerful as thought, and widely held beliefs not as universal as once imagined. Especially true in modern neural networks where results are largely empirical and heuristic and lack the firm footing of many earlier machine learning methods. Because of this, in this review have chosen to emphasize tried and true fundementals, while pointing out what, from current vantage point, seem like promising new techniques.
- In physics, DNNs and CNNs have already found numerous applications:
  - Statistical physics: Detect phase transitions in 2D Ising and Potts models, lattece gauge theories, and different phases of polymers
  - Shown that DNNs can learn free-energy landscapes
  - Methods from statistical physics have been applied to the deep learning field to
    - Study the thermodynamic efficiency of learning rules to explore the hypothesis space that DNNs span,
    - Make analogies between training DNNs and spin glasses, and
    - To characterize phase transitions w/respect to network topology in terms of errors.
  - In relativistic hydrodynamics, deep learning has been shown to capture features of non-linear evolution and hast the potential to accelerate numerical simulations
  - In mechanics CNNs have been used to predict eigenvalues of photonic crystals
  - Recently, DNNs have been used to improve the efficiency of Monte-Carlo algorithms.
  - Deep learning has found interesting applications in quantum physics.
    - Various quanutm phase transitions can be detected and studied using DNNs and CNNs, including
      - The transverse-field Ising model
      - Topological phases
      - And even non-equilibrium many-body localization
    - Representing quantum states as DNNs and quantum state tomography
    - Study quantum and fault-tolerant error correction

* Estimate rates of coherent and incoherent quantum processes
* Recognition of state and charge configurations and auto-tuning in quantum dots
* In quantum information theory, it's been shown one can perform gate-compositions with the help of neural nets
* In lattice quantum chromodynamics, DNNs have been used to learn action parameters in regions of parameter space where PCA fails
* Study of quantum control
* Scattering theory to learn s-wave scattering length of potentials

9.1 Neural network basics

Neural networks/neural nets = neural-inspired nonlinear models for supervised learning. Natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods.

9.1.1 The basic building block: neurons

- The net's basic unit = a stylized "neuron" i that takes a vector of inputs $\boldsymbol{x} = (x_1, x_2, ..., x_d)$ and produces a scalar output $a_i(\boldsymbol{x})$.

- A neural network consists of many such neurons stacked into layers, w/the output of one layer serving as the input of the next.

- First layer = input layer; middle layers= hidden layers; final layer = output layer.

- The exact function a_i varies depending on type of non-linearity used in the neural network. But, in essentially all cases a_i can be decomposed into a linear operation that weights the relative importance of the various inputs and a non-linear transformation $\sigma_i(z)$ which is usually the same for all neurons.

- The linear trans in almost all neural nets takes the form of a dot product with a set of neuron-specific weights $\boldsymbol{w}^{(i)} = (w_i^{(i)}, w_i^{(i)}, ..., w_d^{(i)})$ followed by re-centering with a neuron-specific bias $b^{(i)}$

$$z^{(i)} = \boldsymbol{w}^i \cdot \boldsymbol{x} + b^{(i)} \tag{155}$$

- In terms of $z^{(i)}$ and the non-linear function $\sigma_i(z)$, we can write the full input-output function as

$$a_i(\boldsymbol{x}) = \sigma_i(z^{(i)}) \tag{156}$$

- Historically, common choices of nonlinearities included step-functions (perceptrons), sigmoids (i.e. Fermi functions), and the hyperbolic tangent.
- More recently, has become more common to use rectified linear functions (ReLUs), leaky rectified linear units (leaky ReLUs), and exponential linear units (ELUs).
- Different choices of non-linearities lead to different computational and training properties of neurons. Underlying reason is that we train neural nets using gradient descent based methods that require us to take derivatives of the neural input-output function with respect to the weights $\boldsymbol{w}^{(i)}$ and the bias $b^{(i)}$.
- Notice that the derivatives of the aforementioned non-linearities $\sigma(z)$ have very different properties. The derivative of the perceptron is zero everywhere except where the input is zero. $\rightarrow$ This discontinuous behavior makes it impossible to train perceptrons using gradient descent.
- For this reason, until recently, the most popular choice of non-linearity was the tanh function or a sigmoid/Fermi function. But, this non-linearity choice has a major drawback. When the input weights become large, as they often do in training, the activation function **saturates** and the derivative of the output with respect to the weights tend to zero since $\frac{\partial \sigma}{\partial z} \rightarrow 0$ for $z \gg 1$.
- Such "vanishing gradients" are a feature of any saturating activation function (perceptron $\Theta(z)$, sigmoid $\frac{1}{1+e^{-z}}$, tanh $tanh(z)$), making it harder to train deep networks.
- In contrast, for a non-saturating activation function (such as ReLUs $max(0, z)$, leaky ReLUs ($0.1z$ if $z \leq 0$, z if $z \geq 0$) or ELUs ($e^z - 1$ if $z \leq 0$, z if $z \geq 0$)), the gradients stay finite even for large inputs.

9.1.2 Layering neurons to build deep networks: network architecture

- Basic idea behind all neural networks: layer neurons in a hierarchical fashion, general structure of which is known as the network architecture.
- In the simplest feed-forward networks,
  - each neuron in the *input layer* of the neurons takes the inputs $\boldsymbol{x}$ and produces an output $a_i(\boldsymbol{x} = \sigma_i(z^{(i)}))$ that depends on its current weights.
  - The outputs are then treated as inputs to the next *hidden layer*.

  - This is usually repeated several times until one reaches the top or *output layer*. The output layer is almost always a simple classifier of the form discussed earlier: a logistic regression or soft-max function in the case of categorical data (i.e. discrete labels) or a linear regression layer in the case of continuous outputs.

  Thus, the whole neural net can be thought of as a complicated nonlinear trans of the inputs $\boldsymbol{x}$ into an output $\hat{y}$ that depends on the weights and biases of all the neurons in the input, hidden and output layers.

- Hidden layers greatly expands the representational power of a neural net when compared to a simple soft-max or linear regreesion network.
- Perhaps the most formal expression of the increased representational power of neural networks (=expressivity) is the universal approximation theorem, stating: A neural network with a single hidden layer can approximate any continuous, multi-input/multi-output function with arbitrary accuracy. **The reader is strongly urged to read the beautiful graphical proof of the theorem in free online book (link)**.
  - Basic idea of the proof: Hidden neurons allow neural networks to generate step functions with arbitrary offsets and heights. These can then be added together to approximate arbitrary functions.

  The proof also makes clear that the more complicated a function, the more hidden units (and free parameters) are needed to approximate it. Hence, the applicability of the approximation theorem to practical situations should not be overemphasized.

- In physics, a good analogy are **matrix product states**, which can approximate any quantum many-body state to an arbitrary accuracy, provided the bond dimension can be increased arbitrarily - a severe requirement not met in any practical implementation of the theory.
- Modern neural networks generally contain multiple hidden layers (= the 'deep' in deep learning). There are many ideas why deep architectures favorable for learning.
  - Increasing number of layers = increases the number of parameters = increases the representational power.
  - Recent numerical experiments suggests that as long as the number of parameters is larger than the number of data points, certain classes of neural nets can fit arbitrarily labeled random noise samples. Suggests large neural nets of the kind used in practice can express highly complex functions.

  - Adding hidden layers is also thought to allow neural nets to learn more complex features from the data. Work on CNNs suggests the first few layers learn simple, "low level" features that are then combined into higher-level, more abstract features in the deeper layers.
  - Other works suggest it's computationally and algorithmically easier to train deep netowrks rather than shallow, wider nets, though this is still an area of major controversy and active reasearch (**PAPER**).

- Choosing the exact network architecture remains an art that requires extensive numerical experimentation and intuition, and is often-times problem-specific. Both number of hidden layers and number of neurons in each layer can affect the performance of a neural network. Seems to be no single recipe for the right architecture for a neural net that works best. But, general rule of thumb that seems to be emerging: the numb of parameters in the neural net should be large enough to prevent *underfitting* (**PAPER**).

- Empirically, the best architecture depends on

  - The task
  - The amount and type of data available
  - The computational resources at one's disposal.

  Certain architectures are easier to train, while others might be better at capturing complicated dependencies in the data and learning relevant input features.

- Finally, have been numerous works that move beyond the simple deep, feed-forward neural nets discussed here. Fex: Modern neural nets for image segmentation often incorporate "skip connections" that skip layers of the neural net, allowing information to directly propagate to a hidden or output layer, bypassing intermediate layers and often improving performance.

9.2 Training deep networks

- Basic procedure the same as we used for training simpler supervised learning algos, such as logistic and linear regression: Construct a cost/loss function and then use gradient descent to minimize the cost function. Neural nets differ from these simpler supervised procedures in that generally they contain multiple hidden layers that make taking the gradient more computationally difficult. Will return to this later when discussing the "backpropagation" algo for computing gradients.

- Like all supervised learning procedures, must first specify loss func.
  - Given a data point $(\boldsymbol{x}_i, y_i)$, the neural net makes a prediction$\hat{y}_i(\boldsymbol{w})$, where $\boldsymbol{w}$ are the parameters of the neural network.
  - Recall that in most cases, the top output layer is either a continuous predictor or a classifier. Depending on this one must utilize a different loss function.
  - For continuous data:
    * The loss funcs commonly used are identical to those in linear regression, include the mean squared error

      $$E(\boldsymbol{w}) = \frac{1}{N}\sum_i (y_i - \hat{y}_i(\boldsymbol{w}))^2 \tag{157}$$

      where N is the number of data points, and the mean absolute error (i.e. L_1 norm)

      $$E(\boldsymbol{w}) = \frac{1}{N}\sum_i |y_i - \hat{y}_i(\boldsymbol{w})| \tag{158}$$

      The full cost-func often includes additional terms that implement regularization (e.g. L_1 or L_2 regularizers).
  - For categorical data:
    * Most commonly loss-func is the cross-entropy, since the output layer is often taken to be a logistic classifier for binary data w/two types of labels, or a soft-max classifier if there are more than two types of labels. Cross-entropy already discussed extensively in logistic regression chapter.
    * Recall that for binary data classification, the output of the top layer is the probability $\hat{y}_i(\boldsymbol{w}) = p(y_i = 1|\boldsymbol{x}_i; \boldsymbol{w})$ that datapoint i is predicted to be in category 1.
    * The cross-entropy between the true labels $y_i \in \{0, 1\}$ and the predictions is given by

      $$E(\boldsymbol{w}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i(\boldsymbol{w}) + (1 - y_i)\log[1 - \hat{y}_i(\boldsymbol{w})] \tag{159}$$

    * More generally, for categorical data, y can take on M values so that $y \in \{0, 1, ..., M-1\}$. For each datapoint i, define a vector y_{im} called a 'one-hot' vector, such that

      $$y_i m = \begin{cases} 1, & \text{if } y_i = m \\ 0, & \text{otherwise.} \end{cases}$$

Can also define the prob that the net assigns a datapoint to category m: $\hat{y}_{im}(\boldsymbol{w}) = p(y_i = m|\boldsymbol{x}_i; \boldsymbol{w})$. The categoricla cross-entropy is then

$$E((\boldsymbol{w})) = -\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im}\log\hat{y}_{im}(\boldsymbol{w}) + (1 - y_{im})\log[1 - \hat{y}_{im}(\boldsymbol{w})] \tag{160}$$

As in linear and logistic regr, this loss func often supplemented by additional terms that implement regularization.

- Having identified architecture and cost func, time to train. Use gradient descent based (GD) methods to optimize cost func.
- Recall basic idea of GD = update the parameters $\boldsymbol{w}$ to move in the dir of the gradient of the cost func, $\nabla_{\boldsymbol{w}} E(\boldsymbol{w})$. As discussed there are numerous variants. Most modern NN packages such as Kera allow user to specify which of these optimizers they would like to use in order to train the NN.
- Depending on the architecture, data, computational resources, different optimizers may work better, though vanilla SGD a good first choice.
- Unlike in linear and logistic regression, calc the gradients for a NN requires a special algo= backpropagation = backprop, forming the heart of the NN training.
- Backprop has been discovered multiple times independently, but was popularized for modern NNs in 1985.

9.3 High-level specification of a neural network using Keras

- Load the required packages
- Load the data (the MNIST digit data)
- Data preprocessing:
  - Split between training and testing, reshape
  - Standardize = normalize the input greyscale integer values to values between [0, 1].
  - Encode the classification labels using one-hot vectors, rather than integers (Keras provides function for doing this, "keras.utils.to_categorical")
- Build the network

  - Create an instance of Kera's `Sequential()` class, and call it `model`. Allows us to build DNN's layer by layer. Use `add()` to attach layers to our `model`.
  - We focus on `Dense` layers for simplicity, but in subsequent examples, we'll see how to add dropout regularization and convolutional layers.
  - Every `Dense()` layer accepts as its first required argument an integer specifying the number of neurons.
  - Type of activation for the layer is defined usingt he `activation` optional argument, its input being the name of the activation function, fex `'relu'`, `'tanh'`, `'elu'`, `'sigmoid'`, `'softmax'`.
  - Must ensure number of input and output neurons for each layer match. $\rightarrow$ specify the shape of the input in the first layer of the model explicitly. The sequential construction of the model then allows Keras to infer the correct input/output dimensions of all hidden layers automatically. $\rightarrow$ we only need to specify the size of the softmax output layer to match the number of categories.

- Next, choose loss func to use for training. Chooses `categorical_entropy` defined in Keras' `losses` module (because we're dealing with classification i.e. cross-entropy, and because output data was cast in categorical form).

- Choose SGD for optimization, available in Keras' `optimizers` module. Pass `lr` (learning rate) and `momentum` as optional arguments to the `SGD` func.

- To test performance, look at a particular `metric` of performance. For instance, in categorical tasks typically looks at their `'accuracy'`, which is defined as the percentage of the correctly classified data points.

- Use `compile()` to complete the model, with optional arguments for the `optimizer`, `loss` and the validation `metric`.

- Training the DNN: a one-liner using the `fit()` method of the `Sequential` class. Two first reguired arguments = input and output data. Optional arguments: the mini-`batch_size`, the number of training `epochs`, and the test or `validation_data`. To monitor the training procedure for every epoch, we set `verbose=True`.

9.4 The backpropagation algorithm

Training requires us to calc the derivative of the cost func wrt all the parameters (the weight and biases of all neurons in all layers). A brute force calc is **out of the question** since it requires us to calc as many gradients as parameters at each step of the gradient descent. Backprop = clever procedure that exploits layered structure of NNs to more efficiently compute gradients.

9.4.1 Deriving and implementing the backpropagation equations

- Backprop at its core: The ordinary chain rule for partial differentiation: if z is a function of n variables $x_1, x_2, ..., x_n$ and each of these variables are in turn functions of m variables $t_1, t_2, ..., t_m$. Then for any variable t_i, $i = 1, 2, ..., m$ we have that
$$\frac{\partial z}{\partial t_i} = \frac{\partial z}{\partial x_1}\frac{\partial x_1}{\partial t_i} + \frac{\partial z}{\partial x_2}\frac{\partial x_2}{\partial t_i} + ... + \frac{\partial z}{\partial x_n}\frac{\partial x_n}{\partial t_i} \tag{161}$$
Backprop can be summarized using four equations.

- First: establish useful notation.
  - Assume there are L layers, with $l = 1, ..., L$ indexing the layer.
  - Denote w^l_{jk} the weight for the connection from the k-th neuron in layer $l-1$ to the j-th neuron in layer l.
  - Denote the bias of this neuron by b^l_j.
  - The activation a^l_j of the j-th neuron in the l-th layer can, by construction in a feed-forward NN, be related to the activities of the neurons in the layer $l-1$ by the equation
$$a^l_j = \sigma(\sum_k w^l_{jk} a^{l-1}_k + b^l_j) = \sigma(z^l_j) \tag{162}$$

- We now define the four eqations making up backprop:
  1. By definition, the cost func E depends directly on the activities of the output layer a^L_j (which is our estimated response $\hat{y}$?), but also indirectly on lower layer neuron activations, since a^L_j is calc from them. Define the error of neuron j in layer l, Δ^l_j, as the change in the cost func (why?) wrt the weighted input z^l_j:
$$\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial a^l_j}\sigma'(z^l_j) \tag{163}$$
where $\sigma'(x)$ = the derivative of the non-linearity $\sigma(\cdot)$ wrt its input evaluated at x.
  2. Notice the error func Δ^l_j can also be interpreted as the partial derivative of the cost func wrt the bias b^l_j, since
$$\Delta^l_j = \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial b^l_j}\frac{\partial b^l_j}{\partial z^l_j} = \frac{\partial E}{\partial b^l_j} \tag{164}$$

where we used that $\frac{\partial b_j^l}{\partial z_j^l} = 1$.

3. Derive the final two eq. using the chain rule. Since the error depends on neurons in layer l only through the activation of neurons in the subsequent layer $l+1$, can use chain rule to write

$$\Delta_j^l = \frac{\partial E}{\partial z_j^l} \tag{165}$$

$$= sum_k \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{166}$$

$$= sum_k \Delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{167}$$

$$= (sum_k \Delta_k^{l+1} w_{kj}^{l+1}) \sigma'(z_j^l) \tag{168}$$

4. Derive the final eq by differentiating the cost func wrt w_{jk}^l

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \tag{169}$$

- Together, these are the four backprop equations that relate the gradients of the activations a_j^l, the weighted inputs z_j^l and the errors Δ_j^l. They can be combined into a simple, computationally efficient algo to calc the gradient wrt all parameters: **The Backpropagation Algorithm**

  1. **Activation at input layer**: Calc the activations a_j^1 of all the neurons in the input layer
  2. **Feedforward**: starting w/the first layer, exploit the feed-forward architecture to compute z^l and a^l of each subsequent layer.
  3. **Error at top layer**: calc the error of the top layer using equation 1.
  4. **"Backpropagate" the error**: Use equation 3 to backpropagate the error backwards and calculate Δ_j^l for all layers.
  5. **Calculate gradient**: Use eq. 2 and 4 to calc $\frac{\partial E}{\partial b_j^l}$ and $\frac{\partial E}{\partial w_{jk}^l}$.

- These basic ideas also underly almost all modern automatic differentiation packages such as Autograd (Pytorch).

9.4.2 Computing gradients in deep networks: what can go wrong with backprop?

- Even with backprop, gradient descent on large networks = extremely computationally expensive. The great advances in computational hardware (and the widespread use of GPUs) has made this a much less vexing problem than even a decade ago.

- Gradients can vanish and explode. *The problem of vanishing or exploding gradients.* Especially pronounced in NNs trying to capture long-range dependencies such as RNNs for sequential data. Can illustrate this by considering a simple network w/one neuron in each layer. Assume all weights equal, w. Behavior of the backprop for such a network can be inferred from repeatedly using eq. 3:

$$\Delta_j^1 = \Delta_j^L (w)^L \prod_{j=0}^{L-1} \sigma'(z_j) \tag{170}$$

  Assume the magnitude $\sigma'(z_j)$ fairly constant, so we can approx $\sigma'(z_j) \approx \sigma'_0$. Then, notice that for large L, the error Δ_j^1 has very different behavior depending on the value of $w\sigma'_0$:

  - $w\sigma'_0 > 1$: the errors of the gradient blow up.
  - $w\sigma'_0 < 1$: the errors of the gradient vanish.
  - $w\sigma'_0 \approx 1$ only now, and for neurons not saturated, will the gradient stay well behaved for deep netowrks.

  This behavior holds true even in more complicated networks.

- Rather than considering a single weight, can ask about the eigenvalues (or singular values) of the weight matrices w_{jk}^l. In order for gradients to be finite for deep networks, we need these eigenvalues to stay near unity even after many gradient descent steps.

- In modern feedforward and ReLU NNs, this achieved by initializing the weights for the gradient descent in clever ways, and using nonlinearities that don't saturate, such as ReLUs (recall: for saturating functions, $\sigma' \to 0$, causing the grad to vanish).

- Proper initialization and regularization schemes such as gradient clipping (cutting-off gradients w/very large values), and batch normalization also help mitigate the vanishing and exploding gradient problem.

9.5 Regularizing neural networks and other practical considerations

- DNNs, like all supervised learning algos, must navigate the bias-variance tradeoff.

- Regularization techniques play important role in ensuring DNNs generalize well to new data.

- Last five years seen a wealth of new specialized regularization techniques for DNNs beyond the simple L_1 and L_2 penalties, including

Dropout and Batch Normalization. Also, DNNs seem especially well-suited for implicit reularization that already takes place in the SGD. The implicit stochasticity and local-nature of SGD often prevents over-fitting and spurious correlatios in the training data, especially when techniques such as early stopping. We'll here give a brief overview over these techniques.

9.5.1 Implicit regularization using SGD: initialization, hyper-parameter tuning, and early stopping

- SGD = most commonly employed and effective optimizer for training NNs
- Acts as an implicit regularizer by introducing stochasticity that prevents overfitting.
- Important that weight initialization be chosen randomly, in order to break any leftover symmetries.
  - One common choice: drawing weights from a Gaussian centered around zero with some variance that scales inversley with number of inputs to the neuron. Since SGD a local procedure, as networks get deeper, *choosing a good weight initialization becomes increasingly important* to ensure the gradients are well behaved.
  - Choosing initialization with a variance too large or small causes gradients to vanish and the network to train poorly - even a factor of 2 can make huge difference in practice **PAPER**. $\rightarrow$ **important to experiment with different initialization variances**.
- Second important thing: appropriately choose the learning rate/step size by searching over five logarithmic grid points **PAPER**. If best performance occurs at edge of grid, repeat procedure until optimal learning rate is in the middle of the parameters.
- Common to center or whiten the input data (as we did for linear and logistic regression).
- Early stopping
  - Divide the training data into two portions, the dataset we train on and a smaller validation set that serves as a proxy for out-of-sample performance on the test set.
  - As we train model, we plot both training and validation error. We expect training error to continuously decrease during training. But validation error will eventually increase due to overfitting.

  - Hault the training procedure when validation error starts to rise. This "early stopping" procedure ensures we stop training and avoid overfitting sample specific features in the data.

9.5.2 Dropout

- Basic idea: prevent overfitting by reducing spurious correlations between neurons by introducing a radnomization procedure similar to that underlying ensemble models such as Bagging.
- Recall the basic idea behind ensemble methods: train an ensemble of methods that are created using a randomization procedure to ensure that the members of the ensemble are uncorrelated. This reduces the variance of statistical predictions without creating too much additional bias.
- Extremly costly to train an ensemble of NNs, both due to data needed and computational resources and parameter tuning.
- Droput circumvents this, by randomly dropping out neurons (and their connections) from the NN during each step of the training.
- Typically, for each mini-batch in the grad descent step, a neuron is dropped from the NN with a probability p. The grad descent step then performed only on the weights of the "thinned" network of individual predictors.
- Since in the training, on average weights only present a fraction p of the time, predictions are made reweighing the weights by p: $\boldsymbol{w}_{test} = p\boldsymbol{w}_{train}$.
- The learned weigths can be viewed as some "average" weight over all possible thinned NN. This averaging of weights is similar in spirit to the Bagging procedure discussed in the ensemble methods context.

9.5.3 Batch Normalization

- Basic inspiration: the long-known observation that training in NNs works best when the inputs are centered around zero wrt the bias. Because: it prevents neurons from saturating and gradients from vanishing in deep nets.
- In the absence of such centering, changes in parameters in lower layers can give rise to saturation effects in higher layers, and vanishing gradients.
- Idea: Introduce additional new "BatchNorm" layers that standardize the inputs by the mean and variance of the mini-batch.

- Consider a layer l with d neurons whose inputs are $(z_1^l, ..., z_d^l)$. We standardize each dimension so that

$$\hat{z}_k^l = \frac{z_k^l - E[z_k^l]}{\sqrt{Var[z_k^l]}} \tag{171}$$

where the expectation and variance are taken over all samples in the mini-batch.

- One problem: this may change the representational power of the NN. Fex, for tanh non-linearities, it may force the network to live purely in the linear regime around $z = 0$. Since non-linearities are crucial to the representational power of DNNs, could dramatically alter its power.

- Thus, one introduces two new parameters γ_k^l and β_k^l for each neuron that can shift and scale the normalized input

$$\hat{y}_k^l = \gamma_k^l \hat{z}_k^l + \beta_k^l \tag{172}$$

These new parameters are then learned just like the weights and biases (just an extra layer for the backprop chain rule). We initialize the NN so at the beginning of training the inputs are standardized. Backprop then adjusts γ and β during training.

- In practice, this method considerably improves learning speed by preventing vanishing gradients. But, it also seems to serve as a powerful regularizer for reasons not fully understood. One plausible explanation: in this method, the gradient for a sample depends not only on the sample itself, but also on all the properties of the mini-batch. A single sample can occur in different mini-batches $\rightarrow$ this introduces additional randomness into the training, which seems to help regulize training.

9.6 Deep neural networks in practice: examples

How to use NNs in practice.

9.6.1 Deep learning packages

- There are DNN packages in other languages than Python, fex Caffe in C++.

- Keras = high level framwork, does not require any knowledge about inner workings of the underlying deep learning algos. But, for advanced applications, which may require more direct control over the operations in between layers, Keras' high level design may prove insufficient.

- Keras wraps the functionality of another package - TensorFlow.
- TensorFlow has become the preferred deep learning library. In it one constructs data flow graphs, where nodes=mathematical operations, edges=multidimensional tensors/data arrays. A DNN then thought of as a graph with a particular architecture. One needs to understand this concept well before one can truly unleash TensorFlow's full potential = steep learning curve and requires some time/perseverance.
- There are many other open source packages allowing for control over inter- and intra-layer operations, without need to introduce computational graphs. Fex: Pytorch. Offers libraries for automatic differentiation of tensors at GPU speed. As discussed, mainpulating NNs boils down to fast array multiplication and contraction operations $\rightarrow$ the `torch.nn` library often does the job providing enough access and controllability to manipulate the linear algebra operations underlying DNNs.

9.6.2 Approaching the learning problem

- Typical procedure for using NNs:
  1. **Collect and pre-process the data.**
  2. **Define the model and its architecture.**
  3. **Choose the cost function and the optimizer.**
  4. **Train the model.**
  5. **Evaluate and *study* the model performance on the validation and test data.**
  6. textbfAdjust the hyperparameters (and, if necessary, network architecture) to optimize performance for the specific dataset.
- Step 1: Getting data into appropriate form = inseparable part of the learning process.
  - One of the first questions = how to choose size of training vs test data.
    - MNIST dataset, 10 classification categories: 80% training, 20% testing.
    - ImageNet datset, 100 categories: 50% training, 50% testing.

    Rule of thumb: more classification categories = closer should size of training and test data be, in order to prevent overfitting.
    - Once size of training data set, common to reserve 20% of it for validation, used for fine-tuning the hyperparameters of the model.

  - How to choose the right hyperparameters to begin training with.
    * Fex: according to Bengio, optimal learning rate = an order of magnitude lower than the smallest learning rate that blows up the loss.
    * Also keep in mind, training the model can take a lot of time. This can severely slow down any progress on improving the model in Step 6. $\rightarrow$ usually a good idea to play with a small enough percentage of the training data to get a rough feeling about the correct hyperparameter regimes, the usefulness of the DNN architecture, and to debug one's code. Size of the "play set" should be such that training on it can be done fast and in real time to allow quickly adjusting the hyperparameters.
  - Standardization of the dataset.
    * Shown empirically that if the original values of the data differ by orders of magnitude, training can be slowed/impeded. Related to the vanishing/exploding gradient problem. $\rightarrow$ two tricks here: (i) All data should be mean-centered, i.e. from every data point we subtract the mean of the entire dataset (ii) Rescale the data (ensures the weights of the DNN are of a similar order of magnitude), for which there are two ways: if the data is approximately normally distributed, one can rescale by the standard deviation. Otherwise, it's typically rescaled by the max absolute value so the rescaled data lies in the interval $[-1, 1]$.
  - Often, insufficient data serves as a major bottleneck on the ultimate performance of DNNs. Then one can consider data augmentation, i.e. distorting data samples from the existing dataset in some way to enhance size the dataset. If one knows how to do this, one already has partial information about the important features in the data.

- It's only when steps 1-5 are put together in step 6 that the real benefit of deep learning is revealed, compared to less sophisticated methods such as regression or bagging. The optimal choice of network architecture, cost func, and optimizer is determined by the properties of the training and test data, which are only revealed when we try to improve the model. A typical strategy for exploring the hyperparameter landscape is to use grid searches.

- This procedure can be applied to other ML tasks, not just DNNs. See chapter 11 for more useful hints on how to use the validation data.

9.6.3 SUSY dataset

- Look at the dataset we have previously studied with logistic regression and bagging. There is an interest in using deep learning methods to automatically discover collision features. Benchmark results using Bayesian Decision Trees from a standard physics package, and five-layer NNs using the dropout algorithm were presented in the original paper (reference) to compare the ability of deep learning to bypass the need of using such high-level features (what is meant by this sentence...?).
- Our goal: Study systematically the accuracy of a DNN classifier as a func of the
  - Learning rate
  - Dataset size

  We here use Pytorch.
- We construct a DNN with two dense hidden layers of 200 and 100 neurons, respectively. We use ReLU activation between the input and hidden layers, and a softmax output layer. Apply dropout regularization on the weights of the DNN. Similar to MNIST, we use cross-entropy as cost func and minize it using SGD with batches of size 10% of the training data size. We train the DNN over 10 epochs.
- Fig 40 shows accuracy of DNN on the test data as a func of the learning rate and the size of the dataset.
  - Considered good practice to start with a logarithmic scale to search through the hyperparameters, to get an overall idea for the order of magnitude of the optimal values.
  - In this example, performance peaks at the largest size of the dataset and a learning rate og 0.1, and is of the order of 80%.
  - For comparison, in the original study, the authors achieved $\approx$ 89% by using the entire dataset with 5,000,000 points and a more sophisticated network architecture, trained using GPUs.

9.6.4 Phases of the 2D Ising model

- Study the problem of classifying the states of the 2D Ising model with a DNN, focusing on the model performance as a func of both the number of hidden neurons and the learning rate.
- We construct a minimalistic model for a DNN with a single hidden layer containing a number of hidden neurons. The network architecture thus includes a ReLU-activated input-layer, the hidden layer, and the

softmax output layer. We pick the categorical cross-entropy as cost func and minimize it using SGD with mini-batches of size 100. We train the DNN over 100 epochs.

- Fig 41 show the outcome of the grid search over a log-spaced learning rate and the number of neurons in the hidden layer. See that about 10 neurons are enough at learning rate of 0.1 to get a very high accuracy on the test set. But, if we aim at capturing the physics close to criticality, clearly more neurons are required to reliably learn the more complex correlations in the Ising states.

10 Convolutional Neural Networks (CNN)

- Core lesson in physics: exploit symmetries and invariances. Properties such as locality and translational invariance are often built directly into the physical laws. Our statistical physics model often directly incorporate everything we know about the physical system being analyzed. Fex: we know in many cases it's sufficient to consider only local couplings in our Hamiltonians, or work directly in momentum space if the system is translationally invariant.

- Like physical systems, many datasets and supervised learning tasks also possess additional symmetries and structure. Fex: consider a supervised learning task where we want to label images from some dataset as being pictures of cats or not. Our model must first learn features associated with cats. Because a cat a physical object, we know these features likely to be local (groups of neighboring pixels in the 2D image corresponding to whiskers, tails, eyes, etc.). Also know the cat can be anywhere in the image $\rightarrow$ doesn't really matter where in the picture these features occur (though relative positions of features likely do matter). $=$ a manifistation of translational invariance built into our supervised learning task.

- The all-to-all coupled NNs in the previous section fail to exploit this additional structure. Fex, consider image of the digit 'four' from the MNIST dataset. In the all-to-all coupled NNs used there, the 28×28 image was considered a 1D vector of size $28^2 = 796$. $\rightarrow$ throws away lots of the spatial information.

- The NN community realized these problems, designed a class of NN architecture: CNNs, that take advantage of this additional structure (locality and translational invariance) **PAPER**.

- Interesting for physics: has been recently shown these CNN architectures are intimately related to models such as tensor networks **PAPER** and, in particular, MERA-like architectures that are commonly used in physical models for quantum condensed matter systems **PAPER**.

10.1 The structure of convolutional neural networks

- CNN = translationally invariant NN that respects locality of the input data. Backbone of many modern deep learning applications.
- Reader encouraged to consult notes from the Stanford C231n CNN class this section has been based on
- There are two kinds of basic layers that make up a CNN:
  - A convolution layer that computes the convolution (a mathematical operations, see practical guide to image kernels) of the input with a bank of filters
  - Pooling layers that coarse-grain the input while maintaining locality and spatial structure.

  For 2D data, a layer l is characterized by three numbers: height H_l, width W_l and depth D_l = the number of filters in that layer. All neurons corresponding to a particular filter have the same parameters (i.e. shared weights and bias).
- In general, will be concerned with local spatial filters (often called a receptive field in analogy with neuroscience) that takes as inputs a small spatial patch of the previous layer at all depths. Fex: a square filter of size F = a 3D array of size $F \times F \times D_{l-1}$. The convolution consists of running this filter over all locations in the spatial plane.
- To demonstrate:
  - Consider simple example consisting of a 1D input of depth 1 (fig 43). A filter of size $F \times 1 \times 1$ can be specified by a vector of weights w of length F.
  - The stride, S, encodes by how many neurons we translate the filter by when performing the convolution. In addition, it's common to pad the input with P zeros. For an input of width W, the number of neurons (outputs) in the layer is given by $(W - F + 2P)/S + 1$.
  - See link: visualization of the convolution procedure for a square input of unit depth.
  - After computing the filter, the output is passed through a non-linearity, a ReLU in fig 43. In practice, one often inserts a BatchNorm layer before the non-linearity.

- These convolutional layers are interspersed with pooling layers that coarse-grain spatial information by performing a subsampling at each depth.
  - One common operation = max pool = the spatial dimensions are coarse grained by replacing a small region (say 2×2 neurons) by a single neuron whose output is the max value of the output in the region.
  - In **physics**, this pooling step very similar to the decimation step of RG. This generally reduces the output dimensions.
  - Fex: If region we pool over is 2×2, then both the height and width of the output layer will be halved.
  - Generally, pooling do not reduce the depth of the convolutional layers because pooling is performed seperately at each depth.
  - Some studies suggests pooling might not be necessary, but it remains a staple of most CNNs.
- The convolutional and max-pool layers are generally followed by an all-to-all connected layer and a high level classifier such as soft-max. $\rightarrow$ Allows us to train CNNs as usual using the backprop algo. From a backprop perspective: CNNs almost identical to fully connected NN architectures except with tied parameters.
- Apart from introducing additional structure, such as translational invariance and locality, this conv structure also has important practical and computational benefits.
  - All neurons at given layer represent the same filter $\rightarrow$ can all be described by a single set of weights and biases. Reduces the number of free parameters by a factor of $H \times W$ at each layer.
  - Fex: For a layer with $D = 10^2$ and $H = W = 10^2$, this gives a reduction in parameters of nearly 10^6!
  - $\rightarrow$ Can train much larger models than would otherwise be possible with fully connected layers.
  - Similar phenomena in physics: e.g. in translationally invariant systems we can parametrize all eigenmodes by specifying their momentum (wave number) and functional form (sin, cos, etc), while without translation invariance much more information is required.

10.2 Example: CNNs for the 2D Ising model

- Inclusion of spatial structure = important feature that can be exploited when designing NNs for studying physical systems.

- Used Pytorch to implement a simple CNN composed of
  - A single conv layer followed by a soft-max layer.
  - We varied the depth of the CNN layer from unity - a single set of weights and one bias - to a depth of 50 distinct weights and biases.
  - Trained using SGD for five epochs using a training set consisting of samples from far in the paramagnetic and ordered phases.
- Fig 45 results: T
  - The CNN achieved a 100% accuracy on the test set for all architectures, even for CNN of depth one.
  - Also checked the performance on samples drawn from the near-critical region of temperatures T slightly above and below the critical temperature T_c. The CNN performed admirably even on these critical samples with accuracy between 80% and 90%.
  - As with all ML and NNs, performance on parts of data missing from the training data considerably worse than on test data similar to training data. Highlights the importance of properly constructing an accurate training dataset and the considerable obstacles of generalizing to novel situations.
- Regarding the SUSY dataset, we stress that the absence of spatial locality in the collision features renders applying CNNs to that problem inadequate. (Do they mean that this particular provided input data don't include spatial info, or that spatial info is actually not part of a collision? I'm assuming the first.)

10.3 Pre-trained CNNs and transfer learning

- The immense success of CNNs for image recognition $\rightarrow$ training of huge networks on enormous datasets often by large industrial research teams from Google, Microsoft, Amacon etc. Many of these models known by name: AlexNet, GoogLeNet, ResNet, InceptionNet, VGGNet, etc.
- The trained models have been released, now available in standard packages such as the Torch Vision library in Pytorch, or the Caffe framework. They can be used directly as a basis for fine-tuning in different supervised image recognition tasks through a process called transfer learning.
- Transfer learning basic idea: The filters (receptive fields) learned by the conv layers of these networks should be informative for most image recognition based tasks, not just the ones the were originally trained

for. This turns out to be true in practice for many tasks one might be interested in.

- Three distinct ways one can take a pretrained CNN and repurpose it for a new task:
  - **Use CNN as fixed feature detector at top layer.** If new dataset we want to train on small and similar to original dataset, can simply use the CNN as a fixed feature detector and retrain our classifier = remove the classifier (soft-max) layer at the top of the CNN and replace it with a new classifier (liner SVM or soft-max) relevant to our problem. Here, the CNN serves as a fixed map from images to relevant features (the outputs of the top fully-connected layer right before the original classifier). This prevents overfitting on small, similar datasets. Often useful starting point for transfer learning.
  - **Use CNN as fixed feature detector at intermediate layer.** If dataset small and different from the one used to train the original model, the features at the top level might not be suitable for our dataset. $\rightarrow$ may want to instead use features in the middle of the CNN to train our new classifier. These features thought to be less fine-tuned and more universal (e.g. edge detectors). This is motivated by the idea that CNNs learn increasingly complex features the deeper one goes in the network (see discussion on representational learning in next section)
  - **Fine-tune the CNN.** If dataset large: in addition to replacing and training the classifier in the top layer, can also fine-tune the weights of the original CNN using backprop. May choose to freeze some of the weigths in the CNN during the procedure or retrain all of them simulatneously.
- All this can be carried out easily using packages such as Caffe or the Torch Vision library in PyTorch.

11 High-level concepts in Deep Neural Networks

11.1 Organizing deep learning workflows using the bias-variance tradeoff

- Imagine you're given some data and asked to design an NN for learning how to perform a supervised learning task. What are the best practices for organizing a systematic workflow that allows us to efficiently do this?

- Here we present a simple deep learning workflow inspired by thinking about the bias-variance tradeoff (fig 46). This section draws heavily on tutorial from the Deep Learning School.
- **First thing** we would like to do: divide data in three: training set, validation/development/dev set, and test set. Use validation error as proxy for the test error in order to make tweaks to our model. OBS: do not use any test data to train the algo. This is a cardinal sin in ML.
- **Estimate optimal error rate (Bayes rate)**
  - Establish the difficulty of the task and the best performance one can expect to achieve. No algo can do better than the "signal" in the dataset. Fex: it's likely much easier to classify objects in high resolution images than in blurry, low-resolution ones. $\rightarrow$ Must establish a proxy or baseline for the optimal performance that can be expected from any algo.
  - In Bayesian statistics, this known as the Bayes rate. Since we don't know this *a priori*, we must get an estimate. For many tasks inc speech and object recognition, can approximate this by humans' performance on it. For more specialized task, we would like to ask how well experts, trained at the task, perform. This expert formance then serves as proxy for our Bayes rate.
- **Minimize underfitting (bias) on training data set.**
  - Having the Bayes rate, want to ensure we are using a sufficiently complex model to avoid underfitting on the training data.
  - In practice this means: comparing the training error to the Bayes rate. Since the training error doesn't care about generalization (variance), our model should approach the Bayes rate in the training set.
  - If not, bias of the DNN is too large and should try training the model longer and/or use larger model.
  - If none of these techniques work, likely the model architecture not well suited to the data, should modify the architecture in some way to better reflect the underlying structure of the daya (symmetries, locality, etc.)
- **Make sure you are not overfitting.** Next, run our algo on the validation/dev set.
  - If error similar to training error rate and Bayes rate, we're done.

  - If not, we are overfitting the training data. Possible solutions include: regularization and, importantly, collecting more data.
  - If none of these work: likely has to change the DNN architecture.

- The result
  - If validation and test set drawn from same distributions, good performance on validation set should $\Leftrightarrow$ good performance on the test set. (But typically slightly worse because the hyperparameters were fit to the validation data).
  - But, sometimes training and test data differ in subtle ways. Fex: they're collected using slightly different methods, or it's cheaper to collect data one way vs another. (But why on earth would you then divide training/test data based on this difference? Wouldn't you randomize it?) Rectification: Make two validation/dev sets, one from training data and one from test data. The difference in performance on the two quantifies the train-test mismatch. This can serve as another impoertant diagnostic when using DNNs.

11.2 Why neural networks are so successful: three high-level perspectives on neural networks

As pointed out earlier, the field is rapidly expanding, and many of these perspective may out to be only partially true or even false. Nonetheless, included as guidepost for readers.

11.2.1 Neural networks as representation learning

- Powerful aspect of deep learning: The ability to learn relevant features with relatively little domain knowledge/minimal hand-crafting. Power of deep learning often stems from its ability to act like a black box: take in a large stream of data - find good features capturing the properties of the data we're interested in.

- This ability to learn good representations with very little hand-tuning = one of the most attractive properties of DNNs.

- Many of the other supervised-learning algos discussed (regression-based models, ensemble methods inc random forests or gradient-boosted trees) perform comparably or even better than NNs - but when using hand-crafted features with small-to-intermediate sized datasets.

- The hierarchical structure of deep learning = thought to be crucial to their ability to represent complex, abstract features. Fex: how analysis of CNNs for image classification suggests the lower-levels of the net learn elementary features, such as edge detectors, which are then combined into higher levels of the net into more abstract, higher-level features (e.g. the famous example of a neuron that "learned to respond to cats").

- Has been shown more recently that CNNs can be thought of as **performing tensor decompositions on the data similar to those commonly used in numerical methods in modern quantum condensed matter. PAPER**.
- Interesting consequence of this thinking: One can train a CNN on one large dataset and the features it learn should also be useful for other supervised tasks $\rightarrow$ the ability to learn important and salient features directly from the data, then transfer the knowledge to new task. This ability to learn important, higher-level, coarse grained features is reminiscent of **ideas like the renormalization group (RG) in physics where the RG flows seperate out relevant and irrelevant directions, and certain unsupervised deep learning architectures have a natural interpretation in terms of variational RG schemes. PAPER**.

11.2.2 Neural networks can exploit large amounts of data

- Data explosion
- DNNs are able to exploit the additional signal in large datasets for difficult supervised learning tasks.
- Fundamentally, modern DNNs are unique in that they contain millions of parameters, yet can still be trained on existing hardwares. Complexity of DNNs (in terms of parameters) combined with their simple architecture (layer-wise connections) hit a sweet spot between expressivity (ability to represent very complicated functions) and trainability (ability to learn millions of parameters).
- Indeed, DNN's ability to exploit large datasets thought to differ from many other commonly employed supervised learning methids, fex Support Vector Machines (SVMs). Fig 47: schematic depicting expected performance of DNNs of different sizes with the number of data samples and compares them to supervised learning algos such as SVMs or ensemble methods.
  - When amount of data small: DNNs offer no substantial benefit over these other methods and often performs worse.
  - But: large DNNs seem to be able to exploit additional data in a way other methods cannot.

- Fact that one doesn't have to hand engineer features makes the DNN even more well suited for handling large datasets.
- Recent theoretical results suggest that as long as a DNN large enough, should generalize well and not overfit **ARTICLE**.

11.2.3 Neural networks scale up well computationally

- Modern NNs can harness the immense computational capability that has occurred over the last few decades. Architecture of NNs naturally lends itself to parallelization and the exploitation of fast but specialized processors such as graphical processing units (GPUs).
- Google and NVIDIA set on a course to develop TPUs (tensor processing units) which will be specifically designed for the matematical operations underlying deep learning architectures.
- The layered architecture of NNs also makes it easy to use modern techniques such as automatic differentiation that make it easy to quickly deploy them.
- Algos such as SGD and use of mini-batches make it easy to parallelize code and train much larger DNNs than was thought possible fifteen years ago.
- Many of these computational gains are quickly incorporated into modern packages with industrial resources $\rightarrow$ makes it easy to perform numerical experiments on large datasets, leading to further engineering gains.

11.3 Limitations of supervised learning with deep networks

Supervised learning using NNs has important limitations, like all statistical methods. Especially important when seeking to apply these methods to physics. Often, the same or better performance on a task can be achieved by using a few hand-engineered featuers (or even a collection of random features). Especially important for hard physics problems where data/Monte-Carlo samples maybe hard to come by. Some important limitations:

- **Need labeled data.** Like all supervised learning methods. Can be harder to acquire than unlabeled data (e.g. must pay human experts to label images).
- **Supervised neural networks are extremely data intensive.** Utility of DNNs extremely limited if data is hard to acquire or the datasets are small (hundreds to a few thousands samples). In this case, performance of other methods that utilize hand-engineered features can exceed that of DNNs.

- **Homogeneous data.** Almost all DNNs deal with homogeneous data of one type. Very hard to design architectures that mix and match data types (i.e. some continuous variables, some discrete variables, some time series). In appplications beyond images, video and language, this is often what is required. In contrast, ensemble methods like random forests or gradient-boosted trees have no difficulty handling mixed data types.

- **Many physics problems are not about prediction.** Often not interested in solving prediction tasks such as classification. Want to learn something about the underlying distribution that generates the data. Then, often difficult to cast these ideas in a supervised learning setting. While the problems are related, it's possible to make good predictions with a "wrong" model. The model might or might not be useful for understanding physics.

Some of these remarks particular to DNNs, other shared by all supervised learning methods. Motivates the use of unsupervised methods which in part circumnavigate these problems.

12 Dimensional reduction and data visualization

- Will begin our foray into unsupervised learning by way of data visualization = an important tool in ML to identify structures such as
  - Correlations
  - Invariances (symmetries)
  - Irrelevant features (noise)

  in raw or processed data.

- Conceivably, capturing these properties could help us design better predicitve models. In practice, however, the data we deal with is often high-dimensional = its visualization is impossible - daunting at best.

- Part of the complication = low-dimensional representation of high-dimensional data necessarily incurs information lost.

- Simple way to visualize data: pair-wise correlations (= pairwise scatter plots for all features). Useful in highlighting the important correlations between features when the number of features we are measuring is relatively small.

- In practice, we often have to perform *dimensional reduction* = project the data onto a lower dimensional space = *the latent space.*

- We discuss both linear and non-linear methods for dimensional reduction with applications in data visualization. The techniques can be used in many other applications also, inc lossy data compression and feature extraction.

12.1 Some of the challenges of high-dimensional data

- *High-dimensional data lives near the edge of sample space* Geometry in high-dimensional space can be counterintuitive. Example pertinent to ML:
  - Consider data distributed uniformely at random in a D-dimensional hypercube $\mathcal{C} = [-e/2, e/2]^D$, where $e=$ the edge length.
  - Consider also a D-dimensioanl hypersphere $\mathcal{S}$ of radius $e/2$ centered at the origin and contained within $\mathcal{C}$.
  - The prob that a data point $\boldsymbol{x}$ drawn uniformly at random in $\mathcal{C}$ is contained within $\mathcal{S}$ is well approximated by the ratio of the volume of $\mathcal{S}$ to that of $\mathcal{C}$: $p(||\boldsymbol{x}||_2 < e/2) \sim (1/2)^D$.
  - Thus, as the dimension of the feature space D increases, p goes to zero exponentially fast. $=$ most of the data will concentrate outside the hypersphere, in the corners of the hypercube.
  - In **physics**, this basic observation underlies many properties of ideal gas such as the Maxwell distribution and the equipartition theorem.
- *Real-world data vs. uniform distribution* Fortunately, real-world data is not random or uniformally distributed!
  - Real data usually lives in a much lower dimensional space than the original space in which the features are being measured (see very helpful fig 48).
  - "The blessing of non-uniformity" (vs the curse of dimensionality)
  - Data will typically be locally smooth ($=$ a local variation of the data will not incur a change in the target variable)
  - The idea $=$ similar to statistical physics, where roperties of most systems w/many degrees of freedom can often be characterized by low-dimensional 'order parameters'.
  - In thermodynamics, bulk properties of a gas of weakly interacting particles can be simply described by the thermodynamic variables that enter equation of states rather than astronomically large dynamical variables ($=$ position and momentum) of each particle in the gas is another instantiation of this idea.
- *The crowding problem*

  - When performing dimensional reduction, a common goal is to preserve pairwise distances between the data points from the original space to the latent space.
  - Can be fairly well achieved if the *intrinsic* dimensionality of the data (=that in the original space) is the same as the dimension of the latent space. Again see fig 48 ('rullekake' eksempelet.)
  - But, if one attempts to represent data in a space with dimensionality lower than the intrinsic one, the problem of 'overcrowding' can occur. Means low-dimensional embedding of high-dimensional data are often ambiguous. = two points far apart in the data space are mapped to the vicinity of each other in the latent space.
  - To alleviate this, one needs to weaken the constraint we impose on our visualization schemes. Fex: in the case of t-distributed stochastic embedding (t-SNE), one prioritizes the preservation of short distances or local ordination(?) rather than that of all pairwise distances.

12.2 Principal component analysis (PCA)

- Goal: perform a linear projection of the data onto a lower-dimensional subspace where the variance is maximized. Inspired by the observation that in many cases, relevant information is often contained in the directions with largest variance. (fig 50, helpfull!)
- Intuitively, these dirs encode the large-scale 'signal' as opposed to 'noise' characterized by the dir of small variance.
- PCA also seeks variable dirs while simultaneously reducing the redundancy between new basis vectors. Done by requiring our new basis vectors (=principal components) be orthogonal.
- Data then visualized by projecting it onto a subspace spanned by a few principal component basis vectors.
- Surprisingly, such PCA-based projections often capture a lot of the large scale structure of many datasets. Fex, fig 51 (cool!): shows the projections of samples drawn from the 2D Ising model at various temperatures on the first two principal components. Despite living in a 1600 dimensional space (the samples are 40×40 spins), a single component (i.e. a single direction in this 1600 dimensional space) can capture 50% of the variability contained in our samples. One can actually check that easily that this direction weights all 1600 spins equally and thus **corresponds to the magnetization parameter**. $\rightarrow$ even without any prior physical knowledge, one can extract relevant order parameters using a simple PCA-based projection.

- PCA is widely employed in biological physics when working with high-dimensional data. Recently, **a correspondence between PCA and Renormalization Group flows across the phase transition in the 2D Ising model or in a general setting has been proposed**. In stat phys, PCA has also found application in detecting phase transitions, e.g. in the XY model on frustrated triangular and union jack lattices. Also used to classify dislocation patterns in crystals. Physics has also inspired PCA-based algos to infer relevant features in unlabelled data.

- Concretely,
  - Consider N data points $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ that live in a D-dimensional feature space $\mathbb{R}^D$.
  - Without loss of generality, we assume the empirical mean $\bar{\boldsymbol{x}} = N^{-1}\sum_i \boldsymbol{x}_i$ of these data points is zero (we can always center around the mean: $\bar{\boldsymbol{x}} = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$).
  - Denote $N \times D$ design matrix as $X = [\bar{\boldsymbol{x}}_1, \bar{\boldsymbol{x}}_2, ..., \bar{\boldsymbol{x}}_N]^T$ whose rows = the data points and columns = different features. The $D \times D$ (symmetric) covariance matrix is therefore

    $$\Sigma(X) = \frac{1}{N-1} X^T X \tag{173}$$

    Notice
    * The j-th diagonal entry of $\Sigma(X)$ = the variance of j-th feature
    * The $\Sigma(X)_{ij}$ measures the covariance (i.e. **connected correlation in the language of physics**) between feature i and j.
  - We want: find a new basis for the data that emphasizes highly variable directions while reducing redundancy between basis vectors. In particular, we'll look for **a linear transformation that reduces the covariance between different features**. To do so,
    * First, perform a singular value decomposition (SVD) on design matrix X, namely: $X = USV^T$, where S=a diagonal matrix of singular value s_i, the orthogonal matrix U contains (as its columns) the left singular vectors of X, and similarly V contains (as its columns) the right singular vectors of X.
    * With this, we can rewrite the covariance matrix as

    $$\begin{aligned}\Sigma(X) =& \frac{1}{N-1} VSU^T USV^T && (174)\\ =& V\left(\frac{S^2}{N-1}\right)V^T = & V\Lambda V^T && (175)\end{aligned}$$

where Λ= a diagonal matrix w/eigenvalues λ_i in the decreasing order along the diagonal (i.e. eigendecomposition).

- Clear that the right singluar vectors of X (=columns of V) are principle directions of $\Sigma(X)$, and
- Singular values of X are related to the eigenvalues of covariance matrix $\Sigma(X)$ via $\lambda_i = s_i^2/(N-1)$.
- To reduce the dimensionality of data from D to $\tilde{D} < D$, first construct the $D \times \tilde{D}$ projection matrix $\tilde{V}_{D'}$ by selecting the singular components with the $\tilde{D}$ largest singular values.
- The projection of the data from D to a $\tilde{D}$ dimensional space is simply $\tilde{Y} = X\tilde{V}_{D'}$