2 Mathemacial Preliminaries

2.1 Linear Algebra

- A **tensor** X is an n-dimensional array of elements of the same type. $X \sim (s_1, s_2, \cdot, s_n)$ denotes the shape of the tensor.

2.1.1 Vector Operations

- A property of the dot product is that the maximum value of the dot product of two normalized vectors occurs when both vectors are the same.
  - When $\mathbf{x}$, which represents the input, and $\mathbf{w}$, which represents adaptable parameters, resonate, the dot product is maximized.
  - This is called template matching.

2.1.2 Matrix Operations

- Given two matrices $\mathbf{X}$ and $\mathbf{Y}$, matrix multiplication is defined element wise as: $\mathbf{Z}_{ij} = \mathbf{X}_i \cdot \mathbf{Y}_j$ i.e. the element (i, j) of the product is the dot product of the i-th row of $\mathbf{X}$ and the j-th column of $\mathbf{Y}$.
- The Hadamard method of multiplying matrices is element wise multiplication where each element of the resulting matrix $\mathbf{Z}$ is given by $\mathbf{Z}_{ij} = \mathbf{X}_{ij} \cdot \mathbf{Y}_{ij}$.
- The Hadamard multiplication method is used primarily to mask matrices i.e. setting some elements to zero or scaling operations.
- The Hadamard multiplication method does not preserve linearity and cannot be used in operations where linearity is required. Additionally, it cannot be used in compositions of functions such as $f(g(x))$ because it operates element-wise rather than on the entire structure of the matrices.
- There are many operations that can be done element wise or with whole matrices. PyTorch has built in modules for both types of operations.

2.1.3 Higher-order Tensor Operations

- When in higher dimensions, most of the operations we are interested in are either batched variants matrix operations, or specific combinations of matrix operations and reduction operations.
- Example: with two tensors $\mathbf{X} \sim (n, a, b)$ and $\mathbf{Y} \sim (n, b, c)$, the batched matrix multiplication is defined as $\mathbf{Z} \sim (n, a, c)$ where $\mathbf{Z}_i = \mathbf{X}_i \cdot \mathbf{Y}_i$.

2.2 Gradients & Jacobians

- Gradients play a pivotal role in optimization algorithms by providing semi-automatic mechanisms deriving from gradient descent.

2.2.1 Gradients and Directional Derivatives

- The gradient of a function is defined as:

$$\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} f(\mathbf{x}) \\ \vdots \\ \partial_{x_d} f(\mathbf{x}) \end{bmatrix}$$

- The directional derivative is the dot product of the gradient and the direction vector:

$$\nabla f(x) \cdot \mathbf{v}$$

2.2.2 Jacobians

- Let there be a function $f(x)$ that maps a vector input $\mathbf{x} \sim (d)$ to a vector output $\mathbf{y} \sim (c)$. To calculate the gradient for each output, we must create the **Jacobian** of f.

$$\partial f(\mathbf{x}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_d} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_c}{\partial x_1} & \frac{\partial y_c}{\partial x_2} & \dots & \frac{\partial y_c}{\partial x_d} \end{bmatrix}$$

- Each column of the Jacobian corresponds to the gradient of $f(x)$ that maximizes a specific value within the output vector $\mathbf{y}$.
- Each row of the Jacobian describes how the rate of change for the outputs changes with respect to a specific input.
- When c is equal to 1, i.e. when there is only a single output parameter, the matrix simplifies to a single row vector which is the gradient of the function $f(x)$.
- When $c = 1 = d$, the Jacobian becomes the standard derivative of the function.
- Jacobians inherit the properties of derivatives, including the fact that the Jacobian of a compositions of functions is now the matrix multiplication of the individual Jacobians.
- For a point x_0, the best linear approximation to $f(x)$ is $f(\mathbf{x}_0) + \partial f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0)$. This is called Taylor's theorem.
- A code example:

```
# Generic mathematical function
f = lambda x: x**2 - 1.5*x

# Derivative
df = lambda x: 2*x - 1.5

x = 0.5
f_linearized = lambda h: f(x) + df(x)*(h-x)

#Comparing approximation to actual function
print(f(x + 0.01)) # [Out] = -0.5049
print(f_linearized(x + 0.01)) # [Out] = -0.5050
```

2.3 Numerical Optimization and Gradient Descent

- Consider the problem of trying to find the minimum of a function $f(x)$. Assuming the function has a single output **single-objective optimization**, we try to find a global minimum within an uncontrained domaine.
- It it possible to express the solution in closed-form (where there is a function to find the optimal $\mathbf{x}$), but in general we must resort to iterative procedures.
- Let's start with a random guess $\mathbf{x_0}$ and for every iteration, we decompose the new position as the sum of the old position + the magnitude of the step times the direction of the step:

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \eta_t \cdot \mathbf{p}_t$$

  where η_t is the length of the step and $\mathbf{p}_t$ is the normalized direction vector.
- We call η_t the **learning rate** and a direction $\mathbf{p}_t$ such that $f(\mathbf{x}_t) \leq f(\mathbf{x}_{t-1})$ the **descent direction**.

- Selecting a descent direction for every iteration and being careful with choice of step size will allow us to converge to a local minimum.
- Given that $\mathbf{p}_t$ is the descent direction, it is known that $D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) \leq 0$.
- Given that the directional derivative is the dot product of the gradient and the direction vector, we can conclude:
$$D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) = \nabla f(x_{t-1}) \cdot \mathbf{p}_t = ||\nabla f(x_{t-1})|| \cdot ||\mathbf{p}_t|| \cdot \cos \alpha$$
where α is the angle between the gradient and the descent direction.
- The first term is a constant with respect to $\mathbf{p_t}$, and $||\mathbf{p_t}||$ can be assumed to be equal to 1 as it's a normalized direction vector. With this information, we can simplify the previous formula:
$$D_{\mathbf{p_t}} f(\mathbf{x_{t-1}}) = ||\nabla f(x_{t-1})|| \cdot \cos \alpha$$
- The properties of consine result in it being negative when $\frac{\pi}{2} < \alpha < \frac{3\pi}{2}$, therefore any $\mathbf{p_t}$ that forms an angle a satisfying the previous inequality will be a descent direction.
- The **steepest descent direction** is the direction where $\mathbf{p_t}$ forms an angle of π with $\nabla f(\mathbf{x_{t-1}})$ which is synonymous with $\mathbf{p_t} = -\nabla f(\mathbf{x_{t-1}})$.
- On an intutive level, this makes sense as the gradient points in the direction of greatest increase, so the negative of the gradient would point in the direction of greatest descrease.
- The previous formula can be rewritten as:
$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1})$$
- The step size doesn't matter all that much as long as the size is small enough for f to reduce with each iteration.

2.3.1 Convergence of Gradient Descent

- The formal definition for a local minimum of $f(x)$ is a point $\mathbf{x}^+$ such that the following is true for some $\epsilon > 0$:
$$f(\mathbf{x}^+) \leq f(\mathbf{x}) \; \forall \mathbf{x} : ||\mathbf{x} - \mathbf{x}^+|| < \epsilon$$
- In other words, the function $f(\mathbf{x})$ exists at a local minimum at a point $\mathbf{x}^+$ if for some positive value ϵ, $f(\mathbf{x}^+)$ is less than every point ϵ distance away from $\mathbf{x}^+$.
- By the definition of the local minimum, a function at some local minimum will only ever increase if it enters the neighborhood around the local minimum. Thus the gradient at a local minimum is zero and the gradient around the local minimum is pointing upwards.
- A **stationary point** of $f(\mathbf{x})$ is a point $\mathbf{x}^+$ such that $\nabla f(\mathbf{x}^+) = 0$.
- Stationary points exist at all minima, maxima, and saddle points i.e. where $\nabla f(\mathbf{x}) = 0$.
- Due to this, we can only guarantee that gradient descent will converge to a stationary point, not necessarily a local minimum.
- Ideally, we would want to attain the **global minimum** of a function, the one (or possibly one of many) point(s) in the domain where $f(\mathbf{x})$ attains its lowest possible value.
- For the sake of visualization, assume $f(\mathbf{x}) \in \mathbb{R}^3$. If the function assumes a parabolic shape, then every point in the domain will have a gradient pointing toward the global minimum.

- With the previous example, the topic of **convexity** comes up. A function $f(\mathbf{x})$ is convex if for any two points $\mathbf{x}_1$ and $\mathbf{x}_2$, and $\alpha \in [0, 1]$, we have:

$$f(\underbrace{\alpha \mathbf{x}_1 + (1-\alpha)\mathbf{x}_2}_{\text{Interval from } \mathbf{x}_1 \text{ to } \mathbf{x}_2}) \leq \underbrace{\alpha f(\mathbf{x}_1) + (1-\alpha) f(\mathbf{x}_2)}_{\text{Line segment from } f(\mathbf{x_1}) \text{ to } f(\mathbf{x}_2)}$$

- In words, a function is convex if the line segment connecting two points $f(\mathbf{x}_1)$ and $f(\mathbf{x}_2)$ is always greater than or equal to every single value on the function between $\mathbf{x}_1$ and $\mathbf{x}_2$.
- A convex function simplified our task greatly for the following reasons:
  - For a generic non-convex function, gradient descent will always converge onto a stationary point, not necessarily a local minimum.
  - For a convex function, the stationary point is the global minimum.
  - if the inequality earlier is satisfied in a strict way (**strict convexity**), then the global minimizer is guaranteed to be unique.
- Trying to find the global minimum is a non-convex problem with gradient descent is impossible because you must run the algorithm for an infinite amount of time to check the infinite amount of points from an infinite amount of initializations in the unconstrained domain.

2.3.2 Accelerating Gradient Descent

- A problem with the gradient approach is that it only points to the greatest descent direction in an extemely small neighborhood around the current point. This can lead to very noisy updates and slow convergence.
- To smooth out the erratic changes in descent direction, we can make the direction of the current step to affect the direction of the next step. Such a method is called **momentum**:

$$\mathbf{g}_t = -\underbrace{\eta_t \nabla f(\mathbf{x}_{t-1})}_{\text{gradient descent}} + \underbrace{\lambda \mathbf{g}_{t-1}}_{\text{momentum}}$$
$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{g}_t$$

  where we initialize $\mathbf{g}_0 = 0$ and λ is a parameter that determines how much the previous term is dampened.
- Expanding two terms:

$$\begin{aligned}\mathbf{g}_t &= -\eta \nabla f(\mathbf{x}_{t-1}) + \lambda(-\eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda \mathbf{g}_{t-2}) \\ &= -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}\end{aligned}$$

- The momentum method has been shown to accelerate training by smoothing the optimization path. Also, modifying the step size depending on the gradient is another method. Usually, the step size and the gradient are inversely proportional.

3 Datasets and Losses

3.1 What is a Dataset?

- A supervised dataset S_n of size n is a set of n pairs

$$S_n = \{(x_i, y_i)\}_{i=1}^n$$

  where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an identically and independently distributed draw from some unknown (and unknowable) probability distribution $p(x, y)$.

- A sample being **identicially distributed** means that we are trying to track something that is sufficiently stable in terms of change over time. Take the task of identifying car models from photos. Since car models change over time, we will not be able to to have an identical distribution of car models in a dataset with large discrepancies on the time when the data was collected
- A sample being **independently distributed** means that there is no inherent bias in our training data. This condition would not be satisfied if we exclusively collected data from outside a Tesla dealership.

3.1.1 Variants of Supervised Learning

- In datasets with not enough targets y_i, we can use **unsupervised learning**. Typical applications of unsupervised learning are **clustering algorithms**, where points between clusters are similar and points within clusters are dissimilar. An example of this would be grouping together simlar news articles in terms of topics. Another example is a **retrieval** system, where we retrieve the most similar elements to the user's query.
- Unsupervised learning in itself is not ideal for image classification, because the slightest modifiction to an image can lead to millions of pixels being changed. A model that's already optimized for image classification, a **pre-trained** model, can be used to extract features from the images.
- The states of this model can be interpreted as vectors in a higher-dimensional space. These vectors can be mapped and used to train a classifier.

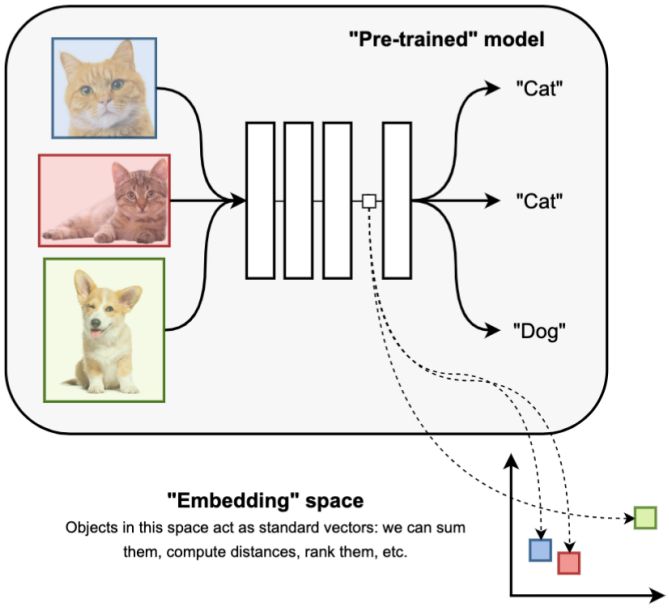

Figure 1: High level overview of using pre-trained model

- **Self-supervised learning** is a variant of un-supervised learning where the model is trained to find some supervised objective from a unsupervised dataset. An example of would be this: A model is given a large piece of text. The model removes a specific part from each sentence in the text and tries to guess the removed part. Comparing its guess to the actual removed part is the supervised objective, the model continuously learns from comparing its guess to the removed part.
- There are three ways of using trained models:
  - **Zero-Shot Learning**: An trained model is given a task it has not training on. It is given no extra data on the task, and must rely on its previous training.

  - **Few-Shot Prompting**: A trained model is given a task it has not training on. It is given a few examples of the task and uses its existing knowledge to make a new inference.
  - **Fine-Tuning**: A trained model is furthet trained on a specific task. This allows it to adapt to the specific needs of the task while using its prior existing knowledge.

- When fine tuning, the model can have all of its parameters changes, or change/add a few parametets. The latter is called **parameter efficient fine-tuning**.
- **Semi-supervised learning** is a variant of supervised learning where the model is trained on a dataset with a small number of labeled examples and a large number of unlabeled examples.

3.2 Loss Functions

- Given a desried targey y and the predicted value $\hat{y} = f(x)$ from a model f, a **loss function** $l(y, \hat{y}) \in \mathbb{R}$ is a scalar, differentiable function whose value correlates with the performance of the model. The performance of the model is measured by the minimization of the loss function i.e. $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ implies that $\hat{y}_1$ is a better prediction than $\hat{y}_2$ wrt the target y.
- The loss function's scalar and differentiable properties allow it to be minimized with a gradient descent algorithm.
- Given a dataset $S_n = \{(x_i, y_i)\}$, and a loss function $l(.,.)$, the optimization task at hand is to minimum average loss on the dataset by any possible differentiable model f:

$$f^* = \arg_f \min \underbrace{\frac{1}{n} \sum_{i=1}^{n}}_{\text{average}} \underbrace{l(y_i, f(x_i))}_{\text{loss value}}$$

- Essentially, we're trying to find the best model that minimizes the loss function. We do this by getting the average of the losses for every single prediction a model makes on a certain dataset. We compare this loss with the average loss of other models on the same dataset, the model with the lowest average loss is best at making predictions for inputs in the dataset.
- This is called **empirircal risk minimization** (risk is generic synonym for loss).
- Models can be parameterized by a set of tensors w (called parameters of the model), and minimization is done by searching for the optimal value of these parameters via numerical optimization, denoted by $f(x, w)$.
- $f(x, w)$ represents the prediction when giving an input x into a model with parameters w.
- Hence, the optimization task can be rewritten as:

$$w^* = \arg_w \min \frac{1}{n} \sum_{i=1}^{n} l(y_1, f(x_1, w))$$

- In this context, we determine the optimal parameters for a specific model that minimizes the average loss on the dataset.

On the differentiability of the loss function

- Consider a model f that outputs a $y \in \{-1, +1\}$ where the true target can only take on two values : -1 and +1.
- We can equate the two possible correct outputs with the sign of f, denoted sign(f(x)).

- One possible loss function is the **0/1 loss**:

$$l(y, \hat{y}) = \begin{cases} 0 \text{ if } \text{sign}(\hat{y}) = y \\ 1 \text{ otherwise} \end{cases}$$

  This is not differentiable, so the gradient descent algorithm would not work to minimize it.

- Another one is **margin** $y\hat{y}$, which will be positive if the prediction is correct and negative otherwise. This is preferable as is is continuously differentiable.

- The **hinge loss** function $l(x, y) = max(0, 1 - y\hat{y})$ is a continuous and differentiable loss function used to train support-vector models.

3.2.1 Expected Risk and Overfitting

- The loss function can be completely minimized if we only respond to inputs already within a dataset, however the objective is to minimize the loss function for all possible inputs.

- The **expected risk** given a probability distribution $p(x, y)$ and a loss function l is defined as:

$$\text{ER}[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))]$$

- This expression shows the expected risk of a model f, denoted $\text{ER}[f]$, for all possible input-output pairs (x, y) on a probability distribution.

- The equation in unfeasiable to calculate, so the **empirical** risk is an estimate of the expected risk with a given dataset.

- The difference in loss between the expected and empirircal risk is called **generalization gap**.

- A overly specific model based on memorization will have a large generalization gap, as it will overfit to the training data, but does not respond well to new data.

- Generalization can be tested by using a separate **test dataset** that the model has not seen before.

3.2.2 Selecting Valid Loss Functions

- Assuming our examples come from a distribution $p(x, y)$, we can decompose it as $p(x, y) = p(x) \cdot p(y|x)$.

- The function $f(x)$ is used to predict $p(y|x)$, that is, the chance that the model will give the correct output y given the input x.

- Approximating $p(y|x)$ with a function $f(x)$ is viable if we assume that the probability mass is mostly centered around a single point y i.e. there are not multiple points $y_1, y_2, \cdots y_n$ that are likely to be be the output.

- However, if we move away from the previous definition of $f(x)$ and instead think of $f(x)$ as a parameterization of the chances of the different outputs $y_1, y_2, \cdots, y_n$ given x, we can represent $f(x)$ as:

$$f(x) = [p(y_1|x), p(y_2|x), p(y_3|x)]$$

- Similarly, we also define $\mathbf{y} \sim \text{Binary}(n)$ where $\mathbf{y}$ is a one-hot encoded vector that contains a 1 at the correct output's place.

- Thus, we can write:

$$p(\mathbf{y}|f(x) = \prod_{i=1}^{3} f_i(x)^{y_1}$$

- This chain of logic can be shown via an example: Assume there is a model, given an input x, outputs a $y \in \{1, 2, 3\}$. The model's chances of giving these outputs are, respectively, $f(x) = [0.2, 0.5, 0.3]$. Assume that the correct output is $y = 2$. Thus, the correct representation of $\mathbf{y} \sim 3$ is $[0, 1, 0]$. Following the expression above, $p(\mathbf{y}|f(x))$ can be calculated as:

$$\begin{aligned} p(\mathbf{y}|f(x)) &= f_1(x)^{y_1} \cdot f_2(x)^{y_2} \cdot f_3(x)^{y_3} \\ &= 0.2^0 \cdot 0.5^1 \cdot 0.3^0 \\ &= 0.5 \end{aligned}$$

  Thus, the probability of the model giving the correct output is 0.5.

- Our formula does not directly give the probability for the output, but instead gives a probability distribution over the possible outputs, with the correct one being one of them.

3.2.3 Maximum Likelihood

- Assume that the samples are independent and identically distributed (i.i.d) from a probability distribution $p(x, y)$. Recall that the probability distribution $p(x, y)$ describes the probability of observing the input x and y together. Hence, the probability assigned to the dataset itself by a specific choice f of function is given by the product of each sample in the dataset:

$$p(\mathcal{S}_n|f) = \prod_{i=1}^{\pi} p(y_i|f(x_i))$$

- This formula defines the probability that a given model f will provide outputs $y_1, y_2, \cdots, y_n$ that are the same as the outputs in the dataset $\mathcal{S}_n$ given the inputs $x_1, x_2, \cdots, x_n$.

- The quantity $p(\mathcal{S}_n|f)$ is called the **likelihood** of the set. This quantity is to be maximized up to a certain extent, where the model's outputs do not deviate greatly from the dataset's outputs but also where the model is also not overfitted with the dataset (and thus struggling with inputs outside of the dataset).

- Given a dataset $\mathcal{S}_n = \{\{x_i, y_i\}\}$ and a family of probability distributions $p(y|f(x)$ parameterized by $f(x)$, the maximum likelihood solution is given by:

$$f^* = \arg_f \max \prod_{i=1}^{\pi} p(y_i|f(x_i))$$

  Essentially, this formula provides the maximum likelihood across all models by providing the likelihood of the best function f^*.

- Switching to a minimization problem, we get:

$$\arg_f \max \left\{ \log \prod_{i=1}^{n} p(y_i|f(x_i)) \right\} = \arg_f \min \left\{ \sum_{i=1}^{n} -\log(p(y_i|f(x_i)) \right\}$$

  A few notes on this equation:

  - We use the log function because multiplying probabilities < 1 can result in a very small product, leading to underflow when storing into memory. Using log helps alleviate this issue.
  - Reminder: the log of a product is equivalent to the sum of the logs of items being multiplied:

$$\log(ab) = \log(a) + \log(b)$$

    Converting this to an addition problem results in reduced computing demand.
  - Adding the negative sign on the right-hand side of the equations turns this into a minimization problem which has been covered earlier.

3.3 Bayesian Learning

- When designing a probability function $p(y|f(X))$ instead of $f(x)$, we can handle situations where the model might give different possible outputs.
- This procedure however only looks an one singular function with a specific parameterization. What if we had multiple functions with different parameterization that all provide good outputs? It would be wasteful to rely on one singular function instead of relying on multiple and choosing the best model for an input based on its output.
- We can achieve this by defining a **prior probability distribution** $p(f)$ over all possible functions. Recall that f is a model with a certain set of parameters.
  - Recall that a **Bayesian prior** is a initial belief on what a parameter might be.
- Functions with smaller norms are preferred so setting up an inverse relationship with the parameters' norm would be useful in creating our priors:

$$p(f) \propto \frac{1}{||f||}$$

- Once a dataset is observed, the probability over f shifts depending on the prior and the likelihood, and the update is given by **Bayes' theorem**.

$$\underbrace{p(f|\mathcal{S}_n)}_{posterior} = \frac{p(\mathcal{S}_n|f)) \cdot \overbrace{p(f)}^{prior}}{p(\mathcal{S}_n)}$$

  The term $p(f|\mathcal{S})$ is called the **posterior distribution function**, while the term $p(\mathcal{S}_n)$ is called the **evidence** and normalizes the right-hand side of the equation.

- Given an input x, we can make a prediction by averaging all possible models based on their posterior's weight:

$$p(y|x) = \int_f p(y|f(x)) \cdot p(f|\mathcal{S}_n) \approx \frac{1}{k} \sum_{i=1}^{k} p(y|f_i(x)) \cdot p(f_i|\mathcal{S}_n)$$

  Here, we take the average of all models when we take the summation and divide by the amount of models on the right hand side.

- If we are solely interested in maximizing the posterior term, we can discard the evidence term as that remains relatively constant. As a result, we can choose only to focus on the terms in the numerator:

$$f^* = \arg\max p(\mathcal{S}_n|f)p(f) = \arg_f \max \left\{ \underbrace{\log p(\mathcal{S}_n|f)}_{likelihood} + \underbrace{\log p(f)}_{regularization} \right\}$$

  This is the **maximum a posteriori** (MAP) solution.

- If all functions have the same weight a priori, then the proble is reduced down to a maximum likelihood solution. The regularizer term pushes the solution toward the basin of attraction defined by the prior distribution.

4 Linear Models

4.1 Least-Squares Regression

4.1.1 Problem setup

Recall that a supervised learning problem can be defined by choosing the input type x, the output type y, the model f, and the loss function l.

- The input is a vector $\mathbf{x} \sim (c)$, corresponding to c number of features in the input.
- The output is a single real value $\in \mathbb{R}$.
  - If y can take any real value, this is a **regression** task/
  - If y can only take one out of m possible values, this is a **classification** task.
  - If y can only take one of two possible values, this is a **binary classification** task.
- We assume f is a linear model.
- We assume that:
  - $n \rightarrow$ size of the database
  - $c \rightarrow$ features of input
  - $m \rightarrow$ classes of outputs

4.1.2 Regression Losses: The Squared Loss and Variants

- Finding the loss for regression is very simple since the prediction error $e = \{\hat{y} - y\}$. However, since we do not concern ourselves with the sign of the loss, rather its absolute value, we can change the loss function to be:
$$l(\hat{y}, y) = (\hat{y} - y)^2$$
- The squared loss function provides many benefits, one of which is that it's rather easy to find the gradient of such a linear function of the model's input.
- Recalling the maximum likelihood principle, the squared loss can be obtained by assuming the outputs of the model follow a Gaussian distribution centered in $f(x)$ with a constance variance of σ^2.
$$p(y|f(\mathbf{x}) = \mathcal{N}(y|f(\mathbf{x})), \sigma^2)$$
Using properties of logs, we can rewrite the log-likelihood as:
$$\log(p(y|f(\mathbf{x}, \sigma^2)) = -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma^2}(y - f(\mathbf{x}))^2$$
- When we minimize for f, the first two terms in the RHS of the equation are constant, leaving the third to be the squared loss. Minimizing for σ^2 is an independent operation and will be touched on later.
- A disadvantage to using the squared loss is that mislabelled points in the source dataset can have an undue effect on the model. Higher errors will be be punished with quadratically growing strenght, which lets **outliers** to negatively impact the model.
- Some loss functions that diminish the influence of outliers are the absolute value loss:
$$l(\hat{y}, y) = |\hat{y} - y|$$
and the **Huber loss**, a combination of the squared loss and the absolute loss:
$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 \text{ if } |y - \hat{y}| \leq 1 \\ (|y - \hat{y}| - \frac{1}{2}) \text{ otherwise} \end{cases}$$
which is quadratic in the proximity of 0 error, and linear otherwise. The $-\frac{1}{2}$ is added for continuity, plot the equations in a 3D grapher to see why.

- Although the absolute value loss function might seem invalid due to its point of non-differentiability, a slight generalization of the derivative called the **subgradient**, can handle this point. On a practical level, gradient descent will never reach the point with perfect precision, so we can assume that derivatives of $|\epsilon|$ for any $\epsilon > 0$ is always defined.

4.1.3 The Least-Squares Model

- A linea model on an input $\mathbf{x}$ is defined as:

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

  where $\mathbf{w} \sim (c)$ and $b \in \mathbb{R}$ (the bias) are trainable parameters.

- The intuition is that given an input $\mathbf{x}$ and parameters $\mathbf{w}$ both holding the same number of features and parameters, each feature of the input will be multiplicated by its respective parameter to produce a number $\in \mathbb{R}$. This number will then be added to the bias b.

- The bias term can be avoided when assuming that a 1 exists as the last feature of $\mathbf{x}$:

$$f\left(\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}\right) = \mathbf{w}^\top \begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \mathbf{w}_{1:c}^\top \mathbf{x} + w_{c+1}$$

- Comibining the linear model, the squared loss, and the empirical risk minimization problem, the **least-squares regression problem** is:

$$\mathbf{w}^*, b^* = \arg_{w,b} \min \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{w}^\top \mathbf{x}_i - b)^2$$

- In code, the linear model can be described as:

```
def linear_model (w: Float[Tensor, "c"],
        b: Float
        x: Float[Tensor, "n␣c"])
        -> Float[Tensor, "n"]:
    return X @ w + B
```

- If we rewrite the least-squares in **vectorized** form, we can achieve optimal computing efficiency:
  The input in vectorized form:

$$\mathbf{X} = \begin{bmatrix}\mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top\end{bmatrix} \sim (n, c)$$

  The output in vectorized form:

$$y = \begin{bmatrix}y_1, \cdots, y_n\end{bmatrix}^\top$$

  The output model for a batch of values is:

$$f(\mathbf{X}) = \mathbf{X}\mathbf{w} + 1b$$

- A note: the ordering of the rows of the input and output do not matter the as the changes will be reflected in $f(\mathbf{X})$. This concept is called **permutation variance** and will be touched on later.

- The vectorized least-squares problem becomes:

$$LS(\mathbf{w}, b) = \frac{1}{n} ||\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}b||^2$$