Training Deep Networks

TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

Training Deep Networks

- We saw a comprehensive description artificial neural networks are.
- We also walked through the **process of forward propagating** information through a network of neurons to output a prediction.
- In those examples, we fabricated numbers for the neuron parameters. In real-world applications, however, these parameters are not typically defined arbitrarily: **they are learned by training the network on data**.

Training Deep Networks

- Now, you will become acquainted with two techniques – **gradient descent** and **backpropagation -** that work in tandem to learn artificial neural network parameters.

- We culminate in the application of these practices to the construction of a neural network with more than one hidden layer.

Cost Functions

- We saw that, upon **forward propagating** some input values all the way through a neural network, it provides its estimated output: $\hat{y}$.
- If a network were perfectly calibrated → it would output $\hat{y}$ values exactly equal to the true label y.
- In our binary classifier for detecting hot dogs:
  - y = 1 indicated that the object presented to the network is a hot dog.
  - y = 0 indicated that it's something else.
- In an instance where we have in fact presented a hot dog → ideally it would output $\hat{y}$ = 1.

Cost Functions

- In practice: the gold standard of $\hat{y} = y$ is not always attained → may be an excessively stringent definition of the "correct" $\hat{y}$.

- Instead, if $y = 1$ we might be quite pleased to see a $\hat{y}$ of 0.9997.
  - This would indicate that the network has an extremely high confidence that the object is a **hot dog**.
  - A $\hat{y}$ of 0.9 might be considered acceptable, $\hat{y} = 0.6$ to be disappointing, and $\hat{y} = 0.1192$ to be awful.

Cost Functions

- A cost function $L(\mathbf{y}, \hat{\mathbf{y}})$ quantifies the prediction error for this single input.

- Typically, we consider the loss **over a large set of observations**, such as the entire dataset.
  - We usually average the losses over all training examples.

- To quantify this spectrum of output-evaluation sentiments, machine learning algorithms often involve **cost functions** (aka **loss functions**).
  - We are going to learn two such functions: **Quadratic cost** and **Cross-entropy cost**.

Quadratic Cost

- Suppose the model has a single continuous-valued output, and (**x**, *y*) is a training example.

- For the same input **x**, suppose the predicted output of the neural net is $\hat{y}$.

- **Quadratic cost** (**squared error loss**) is one of the simplest cost functions to calculate. The cost $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is defined as:

$$L(y, \hat{y}) = (y - \hat{y})^2$$

Quadratic Cost

- In general, we compute the loss for a set of predictions.
- Suppose the **observed** (training set) input-output pairs are:

$$T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

- While the input-ouput pairs **predicted by the model** are:

$$P = \{(x_1, \hat{y}_1), (x_2, \hat{y}_2), \ldots, (x_n, \hat{y}_n)\}$$

Quadratic Cost

- The **Mean Squared Error** (MSE) for the set is:

$$C = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- For any given instance i, we calculate the difference (the error) between the true label y_i and the network's estimated $\hat{y}_i$.
  - Squaring ensures that whether y is greater than $\hat{y}$ or vice versa, the difference between the two is stated as a positive value.
  - Squaring penalizes large differences between y and $\hat{y}$ much more severely than small differences.

Quadratic Cost

- Having obtained a **squared error** for each instance i, we can then calculate the **mean cost C** across all n of our instances by:
  - 1) Summing up cost across all instances;
  - 2) Dividing by however many instances we have.
- Let's see the notebook *Quadratic Cost* to play with this **cost function**.
- This is the function defined to calculate the squared error for an instance i:
- `def squared_error(y, yhat):`
  - `return (y - yhat)**2`

Quadratic Cost

- The mean squared error is just square of the RMSE (**Root Mean Squared Error**).
- **It is convenient to omit the square root to simplify the derivative of the function, which we shall use during training.**
- In any case, when we minimize MSE we also automatically minimize RMSE.

Huber Loss

- **Problem with MSE → it is very sensitive to outliers due to the squared term.**
- A few outliers can contribute very highly to the loss and swamp out the effect of other points.
  - **It makes the training process susceptible to wild swings.**
- One way to deal with this usse is to use **Huber loss**.

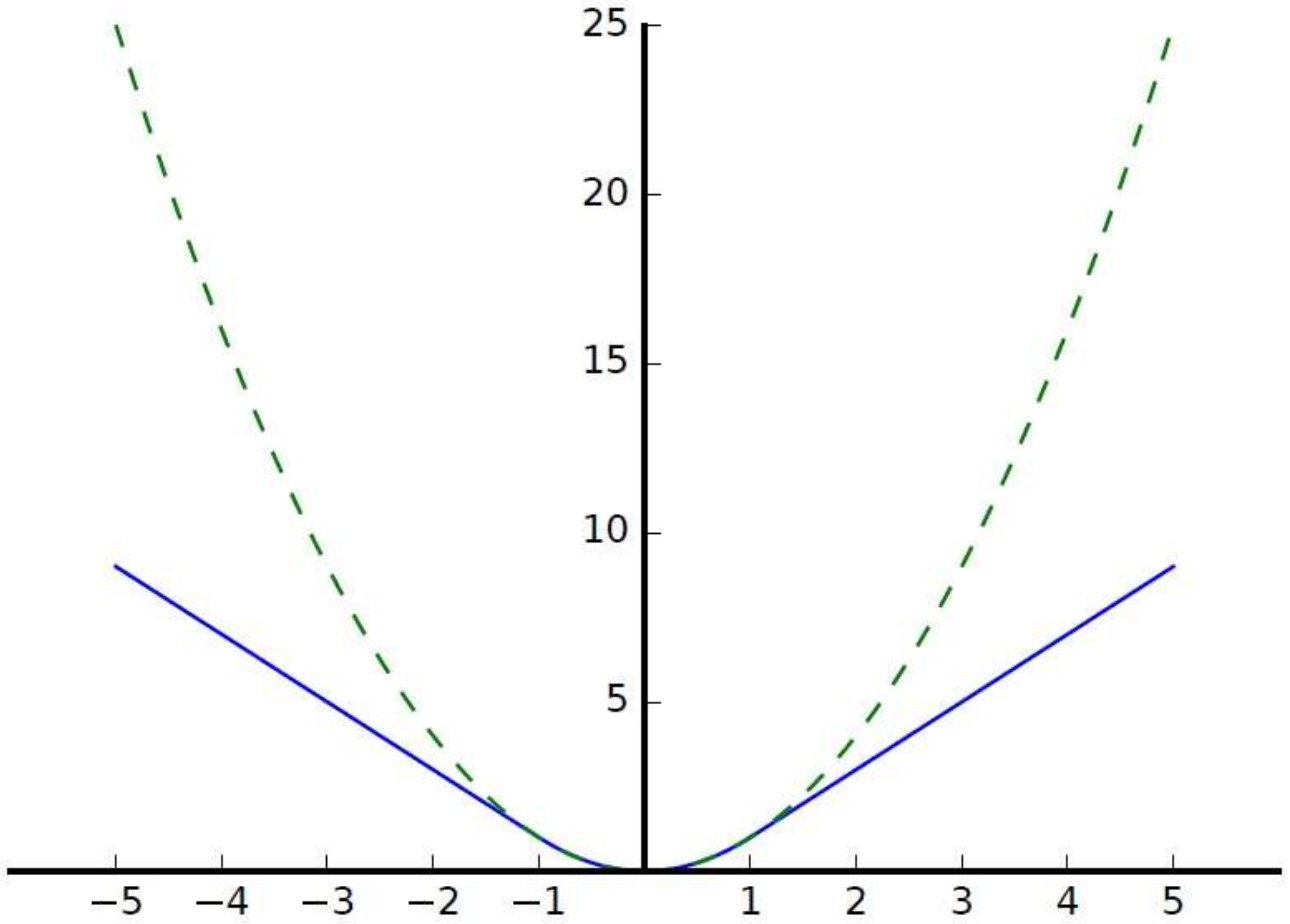
Huber Loss

- Suppose $z = y - \hat{y}$, and δ is a constant.
- The Huber loss L is given by:

$$L_\delta(z) = \begin{cases} z^2 & \text{if } |z| \leq \delta \\ 2\delta(|z| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Huber Loss

- The following figure contrasts the Squared Error and Huber loss functions (Huber Loss - solid line, $\delta = 1$ - and Squared Error - dotted line - as functions of $z = y - \hat{y}$).



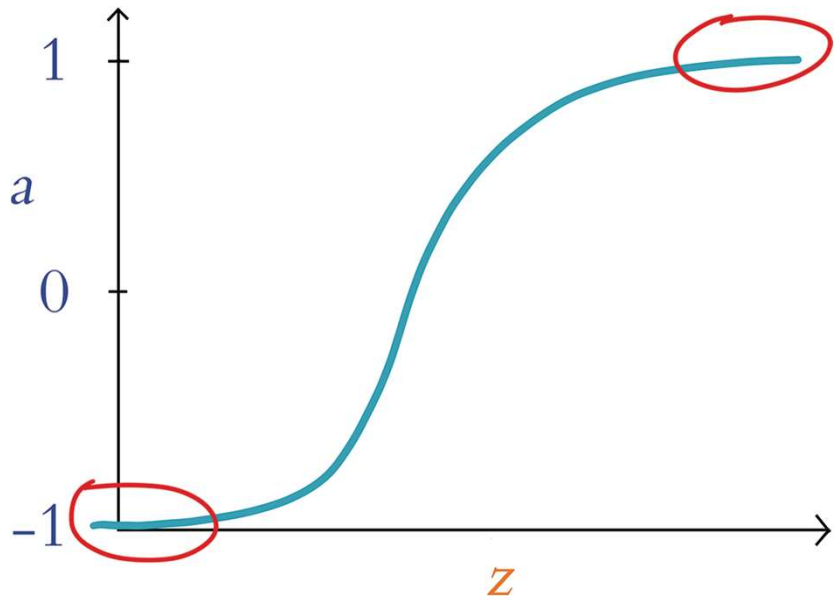
Huber Loss

- In the case where we have a vector $\mathbf{y}$ of outputs rather than a single output, we use $||\mathbf{y} - \hat{\mathbf{y}}||$ in place of $|y - \hat{y}|$ in the definitions of Mean Squared Error and Huber Loss.

Saturated Neurons

- While quadratic cost serves as a straightforward introduction to loss functions, it has a vital flaw.
- This issue, called **neuron saturation**, is common across all activation functions (but we'll use *tanh* as an example).
- A neuron is considered saturated when the combination of its inputs and parameters (interacting in the equation **w . x + b**) produces extreme values of **z.**
- In these areas, changes in **z** cause only tiny in the neuron's activation **a**.

Saturated Neurons



Plot reproducing the tanh activation function, drawing attention to the high and low values of z at which a neuron is saturated.

Saturated Neurons

- Using methods like **gradient descent** and **backpropagation,** a neural network is able to learn to approximate **y** through the tuning of the parameters **w** and **b** associated with all of its neurons.
- In a saturated neuron, where changes to **w** and **b** lead to only minuscule changes in **a**, this learning slows to a crawl.
- These adjustments cannot have any discernible impact downstream (via **forward propagation**) on the network's **ŷ**, its estimate of **y**.

Cross-Entropy Cost

- One of the ways to minimize the impact of saturated neurons on learning speed is to use **cross-entropy cost** in lieu of quadratic cost.
- This alternative loss function is configured to enable **efficient learning** anywhere within the activation function curve.
- Because of this, it is a far more popular choice of cost function and it is the selection that will predominates in the remainder of our classes.
  - In fact, cross-entropy is well suited to neural networks solving classification problems. For regression problems → quadratic cost.
- You need not preoccupy yourself with the equation for **cross-entropy cost**.

Cross-Entropy Cost

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)]$$

Cross-Entropy Cost

- Most pertinent aspects:
- Like quadratic cost, divergence of $\hat{y}$ from y corresponds to increased cost.
- The use of the natural logarithm **ln** in cross-entropy cost causes larger differences between $\hat{y}$ and y to be associated with **exponentially larger cost**.
- Cross-entropy cost is structured so that the larger the difference between $\hat{y}$ and y, the faster the neuron is able to learn.

Cross-Entropy Cost

- As we have seen, comparison of the rate of change of cost **C** relative to neuron parameters like weight **w** is central to **gradient descent** and backpropagation.
  - This **relative rate of change** is given by the partial-derivative $\partial C / \partial w$.
- This cost function is deliberately structured so that, when we calculate its derivative, it is related to $(\hat{y} - y)$.
- Thus, **the larger the difference** between ideal output **y** and the neuron's estimated output $\hat{y}$, **the greater the rate of change** of cost **C** with respect to weight **w**.

Cross-Entropy Cost

- The book presents an analogy to make it easier to understand:
- Let's say you're at a cocktail party leading the conversation of a group of people that you've met that evening.
- The strong martini you're holding has already gone to your head, and you make a bad joke.
- Your audience reacts with visible disgust.
- With this response clearly indicating that your quip was well off the mark, you learn pretty darn quickly.
- It's exceedingly unlikely you'll be repeating the joke anytime soon.

Cross-Entropy Cost

- The final item to note on cross-entropy cost is that, by including $\hat{y}$, it applies to only the output layer.
- One should remember that $\hat{y}$ is a special case of a, since it's being calculated by neurons in the output layer of a neural network.
- With this in mind, the equation could be expressed with a_i substituted in for $\hat{y}_i$ so that the equation generalizes beyond the **output layer** to neurons in **any layer**.

Cross-Entropy Cost

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln a_i + (1 - y_i)\ln(1 - a_i)]$$

Cross-Entropy Cost

- Let's interactively explore our *Cross Entropy Cost* Jupyter notebook.
- There's one defined function:
- ```
  def cross_entropy(y, a):
  ```
  - ```
    return -1*(y*log(a) + (1-y)*log(1-a))
    ```
- We observe comparable behavior to the one carried out by Quadratic Cost.
- These results reiterate for us that the chief **distinction** is the **rate** at which they **learn** within a neural net, especially if saturated neurons are involved.

Cross-Entropy Cost

- Let's see an explanation of the Cross-Entropy Cost.
- Consider a multiclass classification problem with target classes C_1, C_2, …, C_n.
- Suppose each point in the training set is of the form (**x**, **p**)
  - **x** is the input
  - **p** = [p_1, p_2, p_3, …, p_n] is the output.
- p_i is the probability that the input **x** belongs to class C_i, with $\Sigma_i\, p_i = 1$.

Cross-Entropy Cost

- If we are certain that an input belongs to a particular class C_i, then $p_i = 1$ and $p_j = 0$ for $i \neq j$.
- p_i is our level of certainty that input **x** belongs to class C_i, and **p** as a *probability distribution* over the target classes.
- We design our neural network to produce as output a vector:
  - **q** = $[q_1, q_2, q_3, \ldots, q_n]$ of probabilities, with $\Sigma_i\, q_i = 1$.
- **q** is a probability distribution over the target classes, with q_i denoting the model's probability that input **x** belongs to target class C_i.
  - Use of the softmax function in the ouput layer.

Cross-Entropy Cost

- Both the **labeled output** and **the model's output** are probability distributions.
- Which loss function we could use to quantify the distance between both distributions?
- Entropy of a discrete probability distribution **p**:

$$H(\mathbf{p}) = -\sum_{i=1}^{n} p_i \log p_i$$

Cross-Entropy Cost

- Imagine an alphabet of *n* symbols and messages using these symbols.
  - Probability that symbol *i* appears is p_i.
- A key result from **Information Theory**: **if we encode messages using an optimal binary code, the average number of bits per symbol needed to encode messages is *H*(p)**.
- Suppose we we did not know the symbol probability distribution **p** when we design the coding scheme.
- Instead, we believe that symbols appear following a different probability distribution **q**.

Cross-Entropy Cost

- We might ask **what the average number of bits per symbol** will be if we use this suboptimal encoding scheme.

- A well-know result from information theory states that the average number of bits is the **cross entropy** *H(***p***,* **q**):

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{n} p_i \log q_i$$

- Note that *H*(**p**, **p**) = *H*(**p**) and, in general, *H*(**p**, **q**) ≥ *H*(**p**).

Cross-Entropy Cost

- The difference between the cross entropy and entropy is the **average number of additional bits needed per symbol**.

- It is a reasonable measure of the distance between the distributions **p** and **q**, called the **Kullblack-Liebler divergence** (KL-divergence) and denoted D(**p** ll **q**):

$$D(\mathbf{p}\|\mathbf{q}) = H(\mathbf{p},\mathbf{q}) - H(\mathbf{p}) = \sum_{i=1}^{n} p_i \log \frac{p_i}{q_i}$$

Cross-Entropy Cost

- In practice, cross entropy is the most commonly used loss function for classification problems.
- **Networks designed for classification often use a softmax activation function in the output layer.**
- These choices are so common that many implementations, such as TensorFlow, offer a single function that combines softmax with cross entropy.
- **In addition to convenience, one reason to do so is that the combined function is more stable numerically, and its derivative also takes a simple form.**

Cross-Entropy Cost

- KL-divergence is often regarded as a distance, but it is not truly a distance measure because it is not commutative.

- It is perfectly adequate as a loss function for our purposes:
  - There is in fact an inherent assymmetry in the situation.
  - **p** is the ground truth while **q** is the predicted output.

- Notice that minimizing the KL-divergence loss of a model is **equivalent** to minimizing the cross-entropy loss.
  - **Since the term *H(p)* depends only on the input and is independent of the model that is learned.**

Cross-Entropy Cost and Softmax

- Neural networks for classification often use a **softmax activation** in the final layer followed by a **cross-entropy loss**.
- Let's compute the gradient of the combined operator.
- Suppose the input to the combined operator is **y**.
- Let **q** = μ(**y**).
- Let l = H(**p, q**), where **p** represents the true probability vector for the corresponding training example.

Cross-Entropy Cost and Softmax

- We have:

$$\begin{aligned} \log(q_i) &= \log\left(\frac{e^{y_i}}{\sum_j e^{y_j}}\right) \\ &= y_i - \log\left(\sum_j e^{y_j}\right) \end{aligned}$$

Cross-Entropy Cost and Softmax

- Therefore, noting that $\Sigma_i\, p_i = 1$, we have:

$$
\begin{aligned}
l &= H(\mathbf{p}, \mathbf{q}) \\
&= -\sum_i p_i \log q_i \\
&= -\sum_i p_i (y_i - \log(\sum_j e^{y_j})) \\
&= -\sum_i p_i y_i + \log(\sum_j e^{y_j}) \sum_i p_i \\
&= -\sum_i p_i y_i + \log(\sum_j e^{y_j})
\end{aligned}
$$

Cross-Entropy Cost and Softmax

- Diferentiating, we get:

$$\begin{aligned}\frac{\partial l}{\partial y_k} &= -p_k + \frac{e^{y_k}}{\sum_j e^{y_j}} \\ &= -p_k + \mu(y_k) \\ &= q_k - p_k\end{aligned}$$

Cross-Entropy Cost and Softmax

- We end up with the rather neat result:

$$\nabla_{\mathbf{y}} l = \mathbf{q} - \mathbf{p}$$

- This combined gradient **does not saturate or explode**.

- **It leads to good learning behavior.**

- This observation explains why softmax and cross entropy loss work so well together in practice.

Optimization: Learning to Minimize Cost

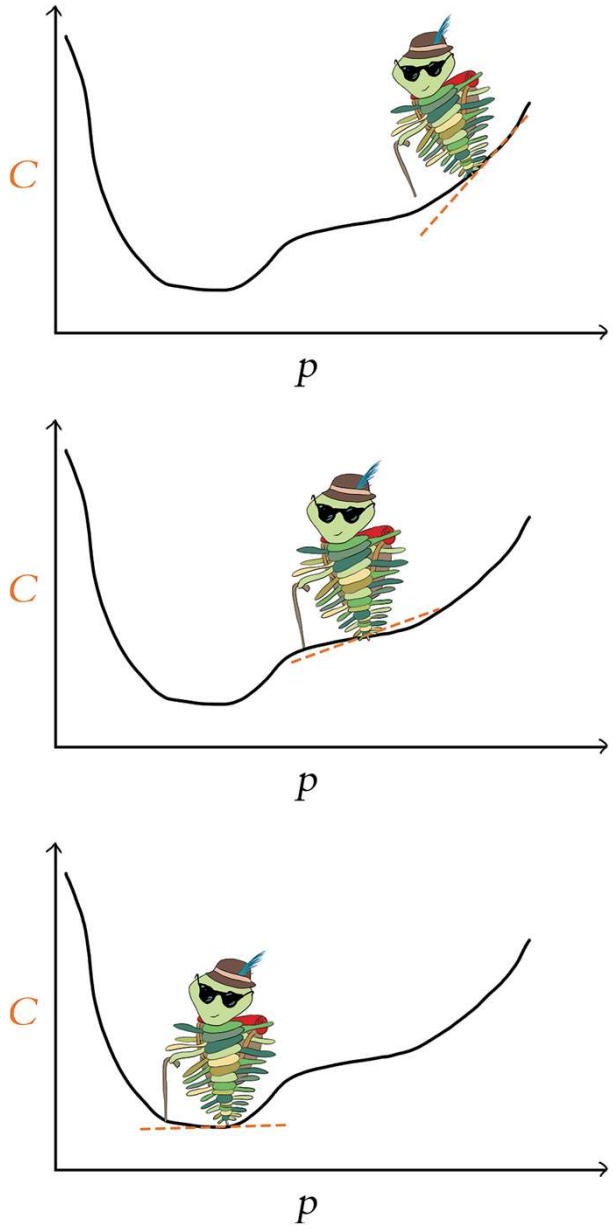
- Cost functions provide us with a quantification of how incorrect our model's estimate of the ideal **y** is → this gives us a metric we can leverage.
- The primary approach for **minimizing cost** in deep learning paradigms: pair **gradient descent** with **backpropagation**.
- These approaches are **optimizers** and they enable the network to **learn**.
- How this learning is accomplished?
  - We adjuste the model's parameters so that its estimated **ŷ** gradually converges toward the target of **y**, and thus the cost decreases.

Gradient Descent

- **Gradient descent** is a handy, efficient tool for adjusting a model's parameters with the aim of minimizing cost, particularly if you have a lot of training data available.

- It is widely used across the **field of machine learning**, not only in deep learning.

- Let's use the nimble trilobite in a cartoon to understand it.



A trilobite using gradient descent to find the value of a parameter ***p*** associated with minimal cost, ***C***.

Gradient Descent

- Along the horizontal axis in each frame is some parameter **p**: in an artificial neural network, this parameter would be either a neuron's weight **w** or bias **b**.

- In the top frame, the trilobite finds itself on a hill. Its goal is to **descend** the gradient → find the location with the minimum cost **C**.

- There's a twist: the trilobite is blind!
  - It can only use its cane to investigate the slope of the terrain in its immediate vicinity.

- The trilobite can then calculate the slope at the point where it finds itself.
  - If the trilobite takes a step to the left (to a slightly lower value of **p**), it would be moving to a location with **smaller cost**.
  - It it takes a step to the right (a slightly higher value of **p**), it would be moving to a location with **higher cost**.

Gradient Descent

- Given the trilobite's desire to **descend the gradient**, it chooses to take a step to the left.
- Later, the trilobite has taken several steps to the left
- Yet again, a step to the left will bring it to a location with **lower cost**.
- It succeeds in making its way to the location - the value of the parameter **p** - corresponding to the minimum cost.
- From this position, if it were to take a step to the left or to the right, **cost would go up**, so it remains in place.

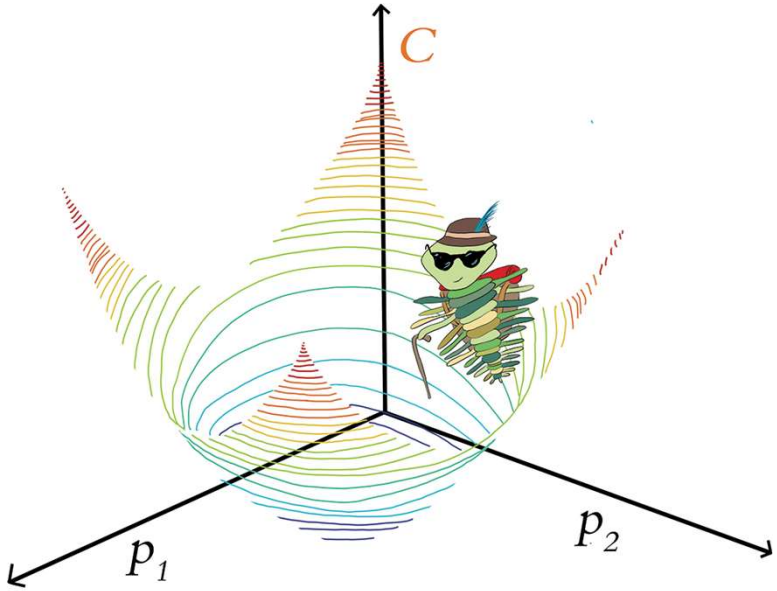
Gradient Descent

- In practice, a deep learning model would not have only one parameter.
- It is not uncommon for deep learning networks to have **millions of parameters**, and some industrial applications have **billions of them**.
- Our *Shallow Net in Keras* has 50,890 parameters.

Gradient Descent

- It's impossible for the human mind to imagine a billion-dimensional space.
- Let's see a two-parameter cartoon that provides a sense of how gradient descent scales up to minimize cost across multiple parameters simultaneously.
- Gradient descent iteratively evaluates slope (using partial-derivatives) to identify the adjustments to those parameters that correspond to the **steepest reduction in cost**.
- With two parameters, for example, this procedure can be likened to a blind hike through the mountains:
  - Latitude represents one parameter: $\mathbf{p_1}$.
  - Longitude represents the other parameter: $\mathbf{p_2}$.
  - Altitude represents cost → **the lower the altitude, the better!**

Gradient Descent

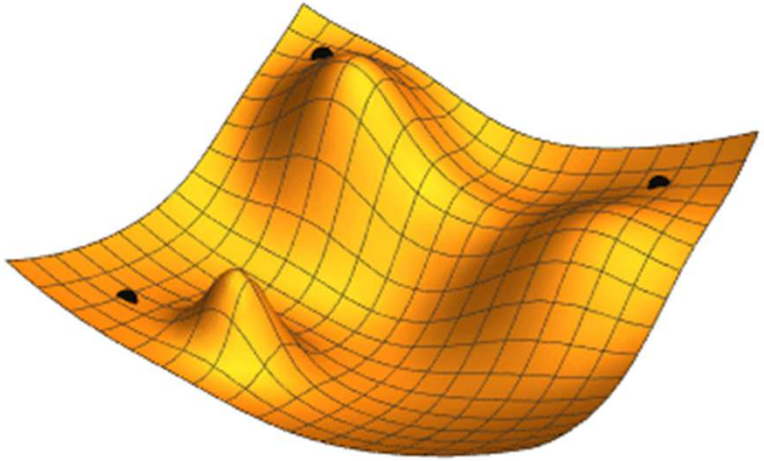


A trilobite exploring along two model parameters—p_1 and p_2—in order to minimize cost via gradient descent.

Gradient Descent

- The trilobite randomly finds itself at a location in the mountains.
- It feels around with its cane to identify the direction of the step it can take that will reduce its altitude the most → It then takes that **single step**.
- Repeating this process many times: the trilobite may eventually find itself at the latitude and longitude coordinates that correspond to the lowest-possible altitude (**the minimum cost**).

Gradient Descent



For three different starting points, varying two parameters so as to minimize the cost function represented by the height.

Learning Rate

- Let's return to a blind trilobite navigating a single-parameter world instead of a two-parameter world.
- Now let's imagine that we have a ray-gun that can shrink or enlarge trilobites.
- In the middle panel, we've used our ray-gun to make our trilobite very small.
- The trilobite's steps will then be correspondingly small, and so it will take our intrepid little hiker a long time to find its way to the valley of minimum cost.
- On the other hand, consider the situtation in which we've used our ray-gun to make the trilobite very large. **The situation here is even worse!**

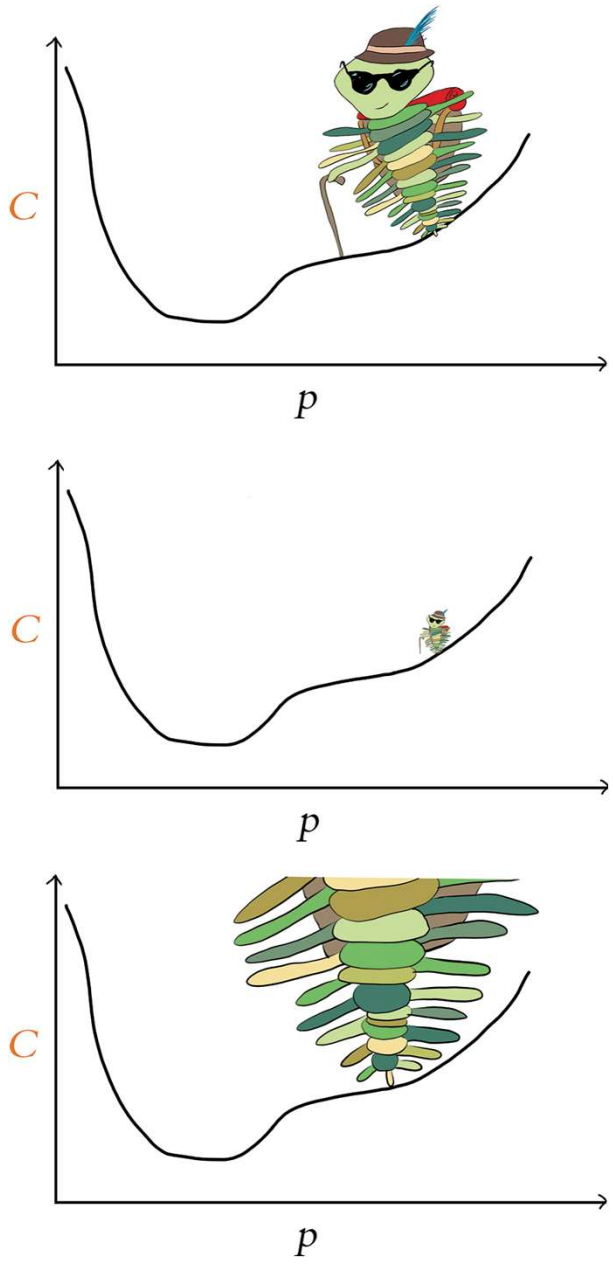
Learning Rate

- The trilobite's steps will now be so large that it will step right over the valley of minimum cost, and so it never has any hope of finding it.

- In gradient descent terminology, step size is referred to as **learning rate** and denoted with the Greek letter η.

- It is our first **hyperparameter**.
  - In machine learning, hyperparameters are aspects of the model that we configure before we begin training the model.

- Hyperparameters are preset, while parameters – **w** and **b** – are learned during training.



The learning rate (η) of gradient descent expressed as the size of a trilobite.

Learning Rate

- Getting your hyperparameters right for a given model → requires some **trial and error**.
- For the learning rate η: too small and too large are both inadequate, but there's a sweet spot in the **middle**.
- If η is too small → **then it will take many, many iterations (unnecessarily) gradient descent to reach the minimal cost**.
- If η is too large → **we might never reach minimal cost at all**.
  - Gradient descent jumps right over the parameters associated with **minimal cost**.

Learning Rate

- We will later see a clever trick that will circumnavigate the need for manually selecting a given neural network's hyperparameter.

- For the interim:
  - Begin with a learning rate of about 0.01 or 0.001.
  - If your model is able to learn (**cost is decreasing**), but training happens very slowly → **increase the learning rate by an order of magnitude (e.g., 0.01 to 0.1)**.
    - If the cost begins to jump up and down erraticaly epoch over epoch → **you've gone too far and should hold back**.
  - If your model is unable to learn, it means that η may be too high → **try decreasing it by orders of magnitude (e.g., from 0.001 to 0.0001) until cost decreases consistently epoch over epoch**.

Learning Rate

- Let's play a little bit in the Neural Network Playground for a visual, interactive way to get a handle on the erratic behavior of a model when its learning rate is too high.

Batch Size and Stochastic Gradient Descent

- If we have a very large quantity of training data, **ordinary gradient descent** would not work at all because it **wouldn't be possible to fit all of the data into the memory** (RAM) of our machine.

- Besides, compute power could cause us headaches too.
  - For a neural network containing millions of parameters, vanilla gradient descent would be highly inefficient because of the **computational complexity of the associated high-volume, high-dimensional calculations**.

- Solution to this: **stochastic gradient descent**.

- We split our training data into **minibatches** - small subsets of our full training dataset - to render gradient descent both manageable and productive.

Batch Size and Stochastic Gradient Descent

- In our *Shallow Network in Keras* notebook, we used used **stochastic gradient descent** by setting our `optimizer` to `SGD` in the `model.compile()` step.

- When we called the `model.fit()` method, we set `batch_size` to `128` to specify the size of our **mini-batches**.

- Like the learning rate, **batch size** is also a model hyperparameter.

Batch Size and Stochastic Gradient Descent

- Let's work through some numbers to make concepts more tangible.
- In the MNIST dataset, there are 60,000 training images.
- With a batch size of 128 images, we then have [468.75] = 469 batches of gradient descent per epoch.

Batch Size and Stochastic Gradient Descent

$$\begin{aligned}\text{number of batches} &= \left\lceil \frac{\text{size of training dataset}}{\text{batch size}} \right\rceil \\ &= \left\lceil \frac{60,000 \text{ images}}{128 \text{ images}} \right\rceil \\ &= \lceil 468.75 \rceil \\ &= 469\end{aligned}$$

Batch Size and Stochastic Gradient Descent

- Before carrying out any training, we initialize our network with random values for each neuron's parameters **w** and **b**.

- To begin the first epoch of training:
  - 1) We shuffle and divide the training images into mini-batches of 128 → It's this shuffling step that puts the stochastic (which means random) in **"stochastic gradient descent"**.

  - Each MNIST image provide 784 pixels each, which all together constitute the inputs **x** that are passed into our neural network.

Batch Size and Stochastic Gradient Descent

- 2) By forward propagation, information about the 128 images is processed by the network, layer through layer, until the output layer produces $\hat{y}$ values.

- 3) A cost function evaluates the network's $\hat{y}$ values against the true y values, providing a cost C for this particular mini-batch of 128 images.

- 4) To minimize cost and thereby improve the network's estimates of y given x, the gradient descent part of stochastic gradient descent is performed:

- Every single w and b parameter in the network is adjusted proportional to how much each contributed to the error (i.e., the cost) in this batch (scaled by the learning rate).

Batch Size and Stochastic Gradient Descent

- These four steps constitute a **round of training**.

Batch Size and Stochastic Gradient Descent

Round of Training:

1. Sample a mini-batch of x values
2. Forward propagate x through network to estimate y with $\hat{y}$
3. Calculate cost C by comparing y and $\hat{y}$
4. Descend gradient of C to adjust w and b, enabling x to better predict y

An individual round of training with stochastic gradient descent. Although mini-batch size is a hyperparameter that can vary, in this particular case, the mini-batch consists of 128 MNIST digits.

Batch Size and Stochastic Gradient Descent

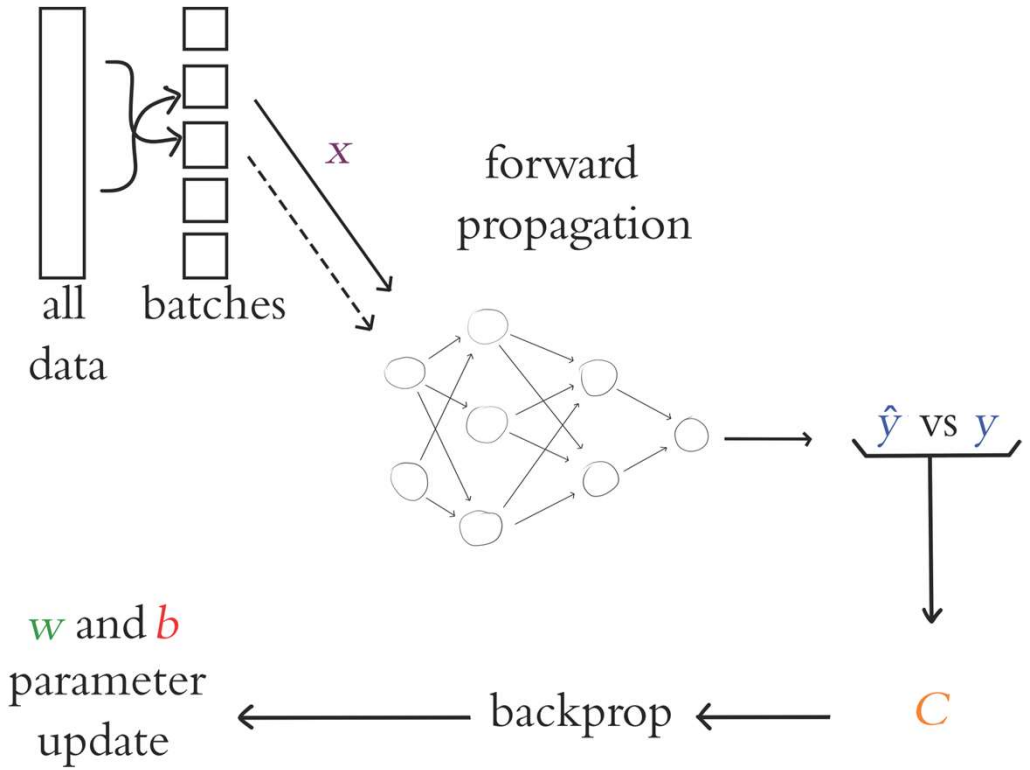
- Rounds of training are repeated until we run out of training images to sample:
  - Note that the sampling in step 1 is done **without replacement**, meaning that at the end of an epoch each image has been seen by the algorithm only once.
  - Between different epochs the mini-batches are sampled randomly.
  - After a total of 468 rounds, the final batch contains only 96 samples.
  - This marks the end of the first epoch of training.
  - Assuming we've set our model up to train for further epochs, we begin the next epoch by replenishing our pool with all 60,000 training images.

Batch Size and Stochastic Gradient Descent

- As we did through the previous epoch, we then proceed through a further 469 rounds of stochastic gradient descent.

- Training continues in this way until the total desired number of epochs is reached.

Batch Size and Stochastic Gradient Descent



An outline of the overall process for training a neural network with stochastic gradient descent.

Batch Size and Stochastic Gradient Descent

- The total **number of epochs** that we set our network to train for is yet another hyperparameter.

- Easy to get right:
  - If the cost on your validation data is **going down epoch over epoch**, and if your final epoch attained the lowest cost yet → **you can try training for additional epochs.**

  - Once the cost on your validation data begins to creep upward → **that's an indicator that your model has begun to overfit to your training data because you've trained for too many epochs.**

  - There are methods you can use to automatically monitor training and validation cost → **stop training early if things start to go awry (set the number of epochs to be arbitrarily large and continue until the validation cost stops improving).**

Local Minima

- **Convex Optimization Problems:** deals with loss functions with a single global minimum.
  - **Simplest case of optimization!**

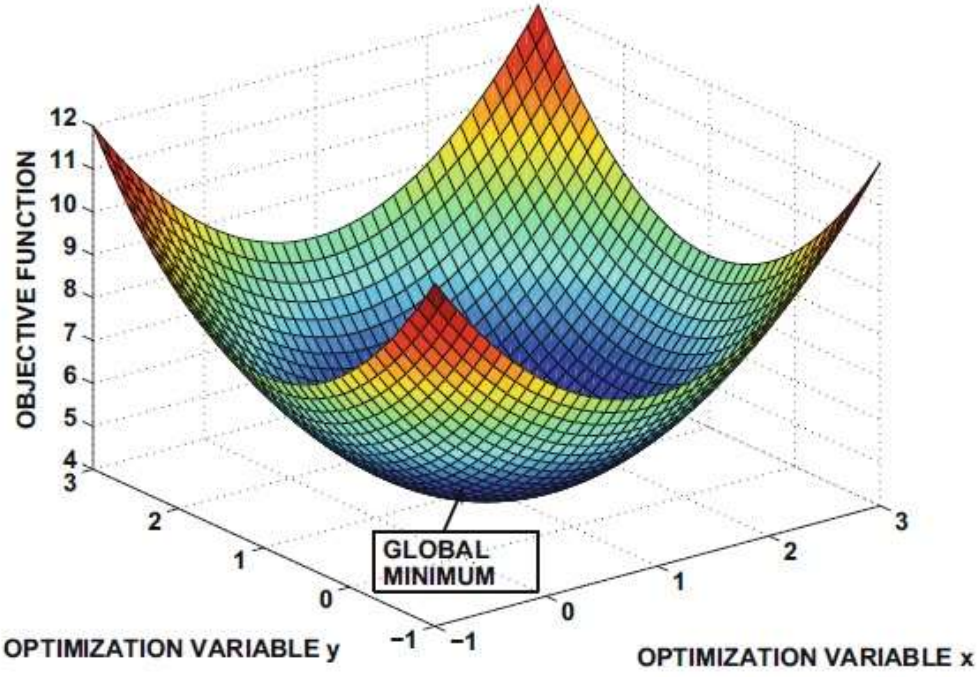
- A loss function $L(\overline{w})$ mapping to real values is said to de **convex**, if it satisfies the following for all weights $\overline{w}_1$ and $\overline{w}_2$, and $\lambda \in (0,1)$:

$$L(\lambda\overline{w}_1 + [1-\lambda]\overline{w}_2) \leq \lambda L(\overline{w}_1) + (1-\lambda)L(\overline{w}_2)$$

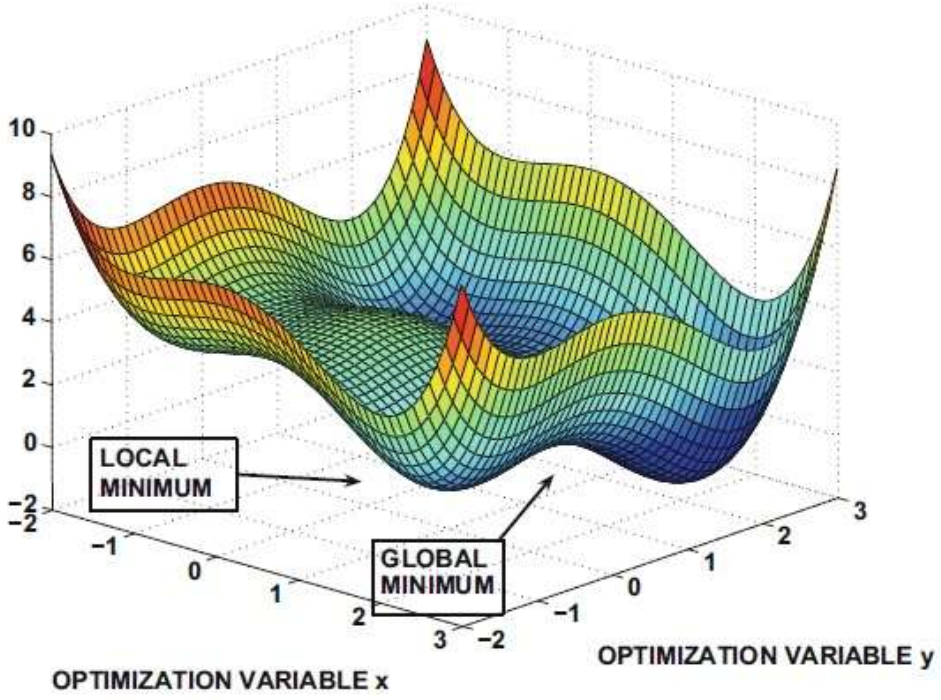
- Examples of convex functions:
  - Constant functions: $f(x) = c$
  - Power of *x*: $f(x) = x^r$ with $r \geq 1$ ($f(x) = x$ is both convex and concave)
  - Reciprocal powers: $f(x) = \frac{1}{x^r}$, for $r > 0$.
  - Logarithm $f(x) = \log x$ is **concave** and exponential $f(x) = e^x$ is **convex**.

Local Minima

- A convex loss function is shaped like is shaped like a bowl with a **single bottom**, whereas a non-convex function might have mutiple “potholes” in the bowl.



(a) Single global minimum
$g(x, y) = x^2 + y^2 - 2x - 2y + 6$



(b) Global and local minimum
$G(x, y) = ([x^4 + y^4]/4)$
$-([x^3 + y^3]/3) - x^2 - y^2 + 4$

Local Minima

- In this example, the function on the left is a convex one and thus has a **single global** minimum at (x, y) = (1, 1).
- The loss function $G(x, y)$ on the right is not convex and **it has multiple local minima**.
- **The convexity of a function is important from the perspective of its behavior during GD.**
- When GD reaches such a pothole, it might be unable to espace from that point: **gradient at the bottom is 0 and all instantaneuous directions of movement worsen the loss function.**

Local Minima

- Most of the shallow networks we have seen have convex loss functions → **this situation does not arise!**
- To prove this, it suffices to show that the second derivative is non-negative.
- General reason: most of these shallow nets can be explained by a **single composition $g(f(.))$ of a convex nonlinear loss function $g(.)$ and a linear function $f(.)$.**
- This composition maintains convexity of the loss function as long **as the convex nonlinear function is applied as the outer nest of this composition**.

Local Minima

- **Example:** the linear function $f(x_1, x_2) = x_1 - x_2$ and quadratic function $g(x) = x^2$ are both convex.
- The function $g\big(f(x_1, x_2)\big) = (x_1 - x_2)^2$ is convex.
  - Let's check this on GeoGebra.
- **However, with increasing depth, the loss function is often not convex.**
- Reason: most activations functions are convex, but composing a linear function (w.x + b) and nonlinear convex fuction (activation) **multiple times (or in the "wrong order") is not always convex.**

Local Minima

- **Example:** the linear function $f(x_1, x_2) = x_1 - x_2$ and quadratic function $g(x) = x^2$ are both convex.

- But the function $f\big(g(x_1), g(x_2)\big) = x_1^2 - x_2^2$ is not convex.
  - Let's check this on GeoGebra.

Local Minima

- A deep neural net is the result of composition of **many linear dot products and nonlinear activation functions**.
- In these cases, the **overall function** computed by the network is **non longer convex in either the inputs or the weight variables.**
- **Consequence:** GD is no longer guaranteed to converge to a global minimum.
- The algorithm can easily get stuck in one of the “potholes” of the loss function, which are called **local optima** (local minima if it’s a minimization problem).

Local Minima

- **Are local minima really a problem?** Only when their objective function values are significantly larger than that of the global minimum.
- **In practice, this does not seem to be the case in neural networks.**
- Many researches have shown that the **local minima of real-life networks have very similar objective function values to the global minimum**.
  - As a result, their presence does not seem to cause a serious problem.
- Some of the training methods that we will learn, such as **momentum methods**, are very well suited to avoiding local optima during GD.

Local Minima

- **Jensen Inequality:** Let $f(x)$ be a convex function on the interval I. Then, $a, b \in I$,

$$f\left(\frac{a+b}{2}\right) \leq \frac{f(a)+f(b)}{2}$$

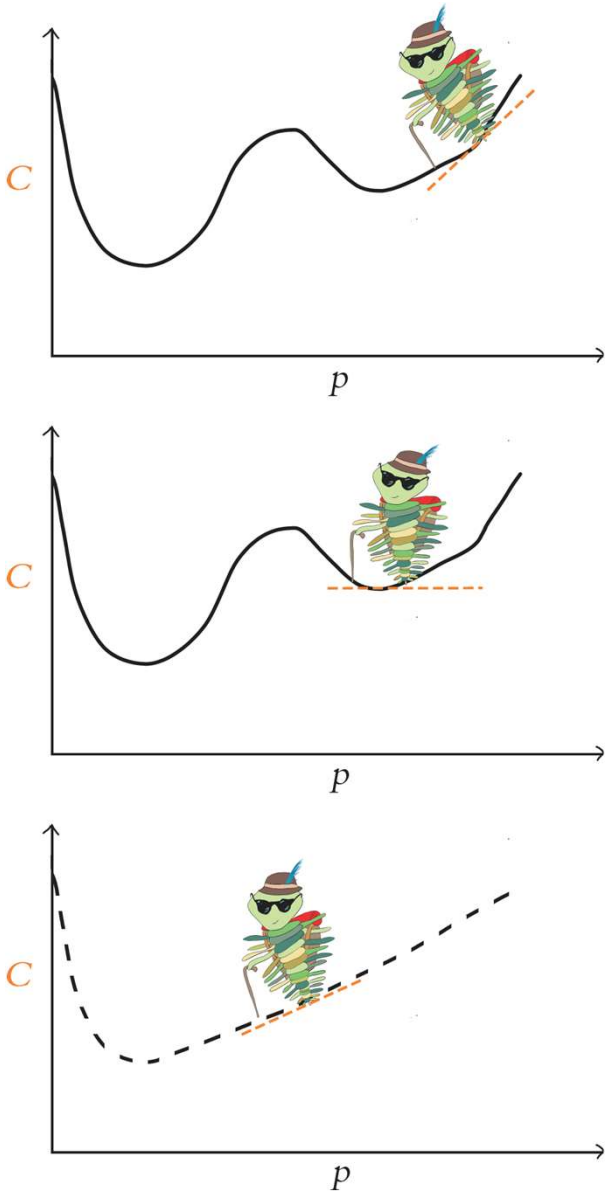
- On the other hand, if $f(x)$ is concave on I, then we have the reverse inequality.

- Interpretation: **for convex *f*, the average of *f* exceeds *f* of the average.**

- **This is a very important inequality that appears in many contexts and problems.**

Escaping Local Minima

- In all of these examples of gradient descent we have seen, our hiking trilobite has encountered no hurdles on its journey toward minimum cost.
  - However, there are no guarantees that this would be the case.
- Let's see an example of the trilobite exploring the cost of some new model that is being used to solve some new problem.
- The relationship between the parameter **p** and cost **C** is more complex.
- To estimate **y** as accuteraly as possible, GD needs to identify the parameter values associated with the lowest-attainable cost.



A trilobite applying vanilla GD from a random starting point gets stuck in a local minimum of cost. Through SGD, the trilobite is able to bypass the local minimum.

Escaping Local Minima

- However, as trilobite makes its way from starting point → GD leads to getting trapped in a **local minimum**.
- In **local minimum**, a step to the left or a step to the right both lead to an increase in cost.
- Trilobite stays put and oblivious of the existence of a deeper valley – a **global minimum**.
- But **Stochastic Gradient Descent** can come to rescue!

Escaping Local Minima

- The sampling of mini-batches can have the effect of smoothing out the cost curve (dashed curve in the drawing).
- This smoothing happens because the estimate is noisier when estimating the gradient for a **smaller mini-batch** (in comparison to the entire dataset).
- Actual gradient in the local minimum truly is zero.
- **Estimates of the gradient from small subsets of the data don't provide the complete picture** and might give an inaccurate reading.
  - It causes our trilobite to take a step left thinking there is a gradient when there really isn't one.
- **This noisiness and inaccuracy is paradoxically a good thing!**

Escaping Local Minima

- Incorrect gradient → step that is large enough for the trilobite to escape the local valley and continue making its way down the mountain.
- By estimating the gradient many times on these mini-batches, the noise is smoothed out and we are able to avoid local minima.
- Each mini-batch on its own lacks complete information about the cost curve → **in the long run this tends to work to our advantage**.

Escaping Local Minima

- Like the learning rate hyperparameter η, there is also a sweet spot for batch size.
- **If the batch size is too large → the estimate of the gradient of the cost function is far more accurate.**
- The trilobite has a more exacting impression of the gradient in its immediate vicinity and is able to take a step in the direction of the **steepest possible descent**.
- However, the model is at risk of becoming trapped in local minima.
- Besides that, the model **might not fit in memory** on your machine, and the **compute time** per iteration of gradient descent **could be very long**.

Escaping Local Minima

- **If the batch size is too small → each gradient estimate may be excessively noisy (very small subset of data is being used).**
- The corresponding path down the mountain will be unnecessarily circuitous.
- Training will take longer because of these erratic gradient descent steps.
- We also are not taking advantage of the memory and compute resources on our machine.
- One shall note that SGD with a batch size of 1 is know as **online learning** (not the fastest method to compute).

Escaping Local Minima

- Rules of thumb for finding the batch-size sweet spot:
  - Start with a batch size of 32.
  - If the mini-batch is too large to fit into memory on your machine, try decreasing your batch size by powers of 2 (e.g., from 32 to 16).
  - If your **model trains well** (cost is going down consistently) but each epoch is **taking very long** and you are aware that you **have RAM to spare** → experiment with increasing your batch size.
    - **To avoid getting trapped in local minima, we don't recommend going beyond 128.**

Exercise

- **Exercise** (4.12.2 from “Neural Networks and Deep Learning: A Textbook”): Consider a 2-feature linear regression problem in which x_1 always lines in the range [8,9] and x_2 always lies in the range [8000,9000]. Comment on why grandient descente is likely to perform more efficiently after feature normalization by inspecting the objective function over a toy tranining set with three examples.

Backpropagation

- SGD operates well on its own to adjust parameters and minimize cost in many types of machine learning models.
- For deep learning: there is an extra hurdle.
- We need to be able to **efficiently adjust parameters through multiple layers** → SGD is partnered up with **backpropagation**.
- **Backpropagation** (or **backprop** for short) is an elegant application of the *chain rule* from calculus.
- **Backpropagation** courses through a neural network in the opposite direction of **forward propagation**.

Backpropagation

- Forward propagation carries information about the input **x** through successive layers of neurons to approximate **y** with **ŷ.**

- **Backpropagation** carries information about the cost **C** **backwards** through the layers in reverse order and adjusts neuron parameters throughout the network (aiming to minimize the cost).

- It's worth understanding what the backpropagation algorithm does:

  - Any given neural network model is randomly initialized with parameter (**w** and **b**) values.
  - Prior to any training, when the first **x** value is fed in, the network outputs a random guess at **ŷ** (**the cost associated with this random guess will probably be high**).

Backpropagation

- We then need to update the weights in order to minimize the cost (essence of machine learning models!).
- We use backpropagation to calculate the **gradient** of the cost function with respect to each weight in the network.
- Recall from our mountaineering analogies: at each step along the way, the trilobite finds the **gradient** (or the slope) of the cost function and moves **down** that gradient → this movement corresponds to a weight update.
- By adjusting the weight in proportion to the cost function's gradient with respect to that weight, **backprop adjusts that weight in a direction that reduces the cost**.

Backpropagation

- From the “most important equation” (**w.x + b**) and through forward propagation through layers → we can grasp that any given weight in the network contributes to the final **ŷ** output, and thus the cost **C**.

- Using backpropagation, we move **layer-by-layer backwards** through the network, starting at the cost in the output layer, and we find the gradients of every single parameter.

- A given parameter’s gradient can then be used to adjust the parameter **up** or **down** (proportional to the learning rate) - whichever of the two directions is associated with a **reduction in cost**.

Backpropagation

- The paper which formalized the Backpropagation algorithm dates to 1986 from David Rumelhart, Geoffrey Hinton and Ronald Williams: *“Learning representations by back-propagating errors”*.
- In 1971, Paul Werbos, during his PhD work, first developed the Backpropagation to mathematicize Freud's "flow of psychic energy".
  - He faced repeated difficulty in publishing the work, only managing in 1981.
- In 1982, Paul Werbos was also the first to apply Backpropagation to ANNs in the way that has become standard.
- This link contains a nice interview with Paul Werbos, where he describes his career, the process of creating Backpropagation, the interactions with Marvin Minsky, as well as many other things.

Backpropagation

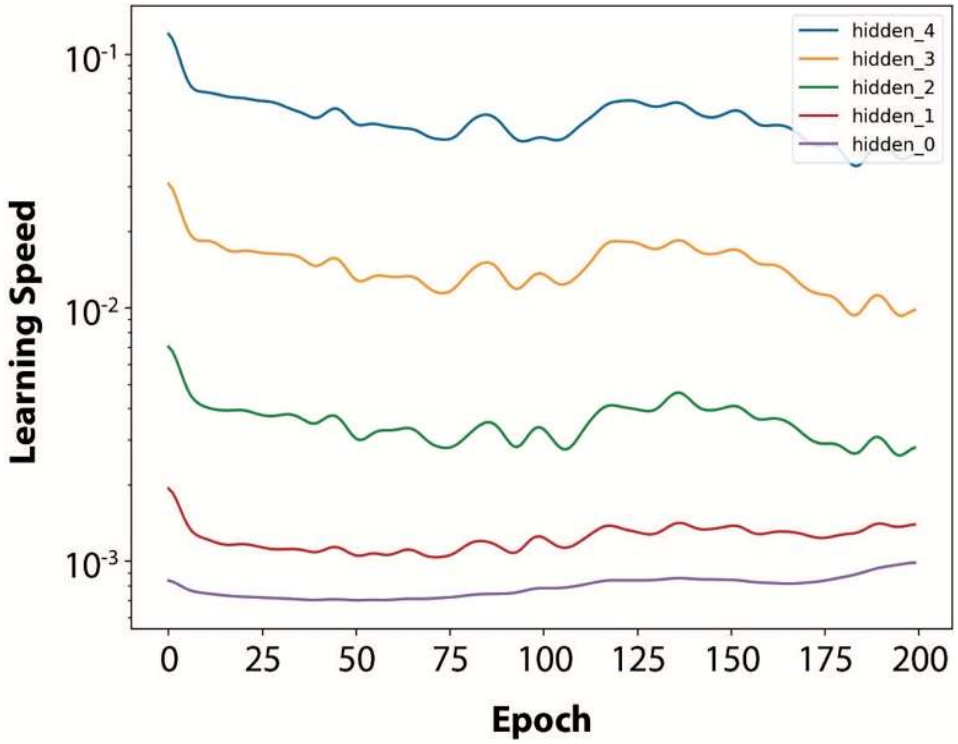
- In sum: **Backpropagation uses cost to calculate the relative contribution by every single parameter to the total cost, and then it updates each parameter accordingly.**

- **In this way, the network iteratively reduces cost and learns!**

Tuning Hidden-Layer Count and Neuron Count

- The number of hidden layers you add to your neural network is also a hyperparameter.
- There's also a sweet spot for your network's count of layers.
- Remember that with each **additional hidden layer** within a DL network, the **more abstract the representations** that the network can represent.
  - That is the primary advantage of adding layers.
- The disadvantage of adding layers is that backpropagation becomes less effective:
  - Backprop is able to have its greatest impact on the parameters of the hidden layer of neurons closest to the output $\hat{y}$.
  - The farther a layer is from $\hat{y}$, the more diluted the effect of that layer's parameters on the overall cost.

Tuning Hidden-Layer Count and Neuron Count



The speed of learning over epochs of training for a deep learning network with five hidden layers.

Tuning Hidden-Layer Count and Neuron Count

- The fifth layer, closest to the output $\hat{y}$, learns **most rapidly** because those weights are associated with larger gradients.
- The third hidden layer learns about **an order of magnitude more slowly** than the fifth hidden layer.
- The first layer is **several layers away from the output layer's cost calculation**.
  - It learns about **two orders of magnitude more slowly** than the last one.

Tuning Hidden-Layer Count and Neuron Count

- Rules of thumb for selecting the number of hidden layers:
  - The more abstract the ground-truth value **y** you'd like to estimate with your network, the more helpful additional hidden layers may be. **One shall start off with about two to four hidden layers.**
  - If reducing the number of layers does not increase the cost you can achieve on your validation dataset, then do it → **Occam's razor**; it will train more quickly and require fewer compute resources.
  - On the other hand, if increasing the number of layers decreases the validation cost, then you should pile up those layers!

Tuning Hidden-Layer Count and Neuron Count

- The number of neurons in a layer is a hyperparameter too.
- If you have many layers in your network, then there are many layers you could be fine-tuning your neuron count in.
- A few too many neurons, and your network will have a touch **more computational complexity than is necessary**.
- A touch too few neurons, and your network's accuracy **may be held back imperceptibly**.

Tuning Hidden-Layer Count and Neuron Count

- With experience, you'll begin to develop a sense for how many neurons might be appropriate in a given layer.
- **Depending on the particular data you're modeling**, there may be lots of low-level features to represent, in which case you might want to have more neurons in the network's early layers.
- If there are lots of higher-level features to represent, then you may benefit from having additional neurons in its later layers.
- To determine this empirically, **we generally experiment with the neuron count in a given layer by varying it by powers of 2**.

Tuning Hidden-Layer Count and Neuron Count

- If doubling the number of neurons from 64 to 128 provides an appreciable improvement in model accuracy, then go for it.

- **Rehashing Occam's razor**: if halving the number of neurons from 64 to 32 doesn't detract from model accuracy → that's probably the way to go because you're reducing your **model's computational complexity** with no **apparent negative effects**.

Occam's razor

- **Occam's razor:** Prefer the simplest hypothesis that fits the data.
- William of Occam was one of the first to discuss this question, Around the year 1320.
- Principle of parsimony: "Entities must not be multiplied beyond necessity" (translated from latin).
- Albert Enstein formulation: "Everything should be made as simple as possible, but not simpler."
- **The explanation of a series of observations must be as simple as possible, without resorting to superfluous concepts.**

Occam's razor

- Scientists sometimes appear to follow this inductive bias.
  - As few as possible: equations, hypothesis and free parameters.
  - Physicists → simple explanations for the motions of the planets.

- Is this **inductive bias favoring shorter hypothesis** a sound basis for generalizing beyond the training data?
  - Philosophers and others have debated this question for centuries, and this remains unresolved to this day.

- Epistemologist Karl Popper defined the quality of a theory by its ability to predict, rather than by its ability to explain existing observations.
  - **Training data accuracy x test data accuracy.**

Occam's razor

- He defines the scientific method as a procedure in which theories must be "**falsifiable**" to earn the label "**scientific**".

- **An overly complex theory can always be adjusted to explain new observations → it cannot be falsified.**
  - Like a polynomial of top degree that can always be adjusted to pass through a new point.

- **A more economical theory is not as easy to adjust → new data can confirm or invalidate it. It is falsiable!**

Occam's razor

- **One argument:** there are fewer short hypothesis than long ones (based on straightforward combinatorial arguments).
- **It is less likely that one will find a short hypothesis that coincidentally fits the training data.**
  - It is less likely to be a statistical coincidence and thus we should prefer this hypothesis.
- **In contrast, there are often many very complex hypothesis that fit the current trainig data but fail to generalize correctly to subsequent data.**
- Decision tree: many more 500-node trees than 5-node trees!

An Intermediate Net in Keras

- Using all of this knowledge that we've learned, let's take a look at the *Intermediate Net in Keras* notebook.
- When we design our neural network, we have used the `relu` activation function instead of `sigmoid`.
- Other significant change → we specify a second hidden layer of artificial neurons.
- By calling the `model.add()` method, we nearly effortlessly add a second `Dense` layer of `64 relu` neurons → an intermediate-depth neural network.

An Intermediate Net in Keras

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

A summary of the model object from our *Intermediate Net in Keras* Jupyter notebook.

An Intermediate Net in Keras

- In this shallow architecture, we specify a second hidden layer of neurons.
- We also set our loss function to cross-entropy cost by using `loss='categorical_crossentropy'`.
- We specify our SGD learning rate hyperparameter by setting `lr=0.1`.

An Intermediate Net in Keras

- We have also reduced our epochs hyperparameter from `200` down by an order of magnitude to `20`.
- Our much-more-efficient intermediate architecture **required far fewer epochs to train**.
- Recall that our shallow architecture plateaued as it approached 86% accuracy on the validation dataset after 200 epochs.
- Our intermediate-depth network is clearly superior:
  - The `val_acc` field shows that we attained 92.34% accuracy **after a single epoch of training**.
  - This accuracy climbs to more than 95% by the third epoch and appears to plateau around 97.6 percent by the twentieth.

An Intermediate Net in Keras

```
Epoch 1/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.4744 - acc: 0.8637 - val_loss: 0.2686 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.2414 - acc: 0.9289 - val_loss: 0.2004 - val_acc: 0.9404
Epoch 3/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1871 - acc: 0.9452 - val_loss: 0.1578 - val_acc: 0.9521
Epoch 4/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1538 - acc: 0.9551 - val_loss: 0.1435 - val_acc: 0.9574
```

The performance of our intermediate-depth neural network over its first four epochs of training

An Intermediate Net in Keras

- We have also reduced our epochs hyperparameter from `200` down by an order of magnitude to `20`.
- Our much-more-efficient intermediate architecture **required far fewer epochs to train**.
- Recall that our shallow architecture plateaued as it approached 86% accuracy on the validation dataset after 200 epochs.
- Our intermediate-depth network is clearly superior:
  - The `val_acc` field shows that we attained 92.34% accuracy **after a single epoch of training**.
  - This accuracy climbs to more than 95% by the third epoch and appears to plateau around 97.6 percent by the twentieth.

Referências

1. Capítulo 8 do livro *"Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence"* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.

2. Capítulo 13 do livro *"Mining of Massive Datasets"* de Jure Leskovec, Anand Rajaraman e Jeffrey Ullman.

3. Capítulo 4 do livro "*Neural Networks and Deep Learning: A Textbook"* de Charu C. Aggarwal. 2ª ed.

4. Capítulo 3 do livro *"Machine Learning"* de Tom Mitchell.

Referências

5. Capítulo 9 do livro "*Quand la machine apprend: La révolution des neurones artificiels et de l'apprentissage profond*" de Yann Le Cun.