Deep Learning

04 - Model Training. Fitting & Backprop

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri

Fitting Models

(Chapter 6)

SAPIENZA
UNIVERSITÀ DI ROMA

Recapping

- The loss we saw in the previous class depends on the network parameters, and here we show how to find the parameter values that minimize this loss.
  - This is known as learning the network's parameters or simply as training or fitting the model.
- The process is to choose initial parameter values and then iterate the following two steps:
  - compute the derivatives (gradients) of the loss with respect to the parameters
  - adjust the parameters based on the gradients to decrease the loss.
- After many iterations, we ***hope*** to reach the overall minimum of the loss function.
- We will show algorithms that adjust the parameters to decrease the loss.

Gradient Descent

- Starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates two steps:

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\mathrm{L}[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i \qquad \frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}.$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \frac{\partial L}{\partial \boldsymbol{\phi}},$$

Learning Rate

where the positive scalar α determines the magnitude of the change.

Colab time: Line Search

https://colab.research.google.com/drive/1mxj-tPKkdLtLWfJnvF1DhFrvwcDZADXD

Gradient Descent: Linear Regression Example

- The model $y = f[x,\boldsymbol{\phi}]$ maps a scalar input x to a scalar output y and has parameters $\boldsymbol{\phi} = [\phi_0, \phi_1]^T$, which represent the y-intercept and the slope:

$$\begin{aligned} y &= f[x, \boldsymbol{\phi}] \\ &= \phi_0 + \phi_1 x \end{aligned}$$

- Given a dataset $\{x_i, y_i\}$ containing I input/outp ast squares loss function:

$\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$ is the individual contribution to the loss from the i^{th} training example

$$\begin{aligned} L[\boldsymbol{\phi}] &= \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} (f[x_i, \boldsymbol{\phi}] - y_i)^2 \\ &= \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2 \end{aligned}$$

Gradient Descent: Linear Regression Example

- The derivative of the loss function with respect to the parameters can be decomposed into the sum of the derivatives of the individual contributions:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \frac{\partial}{\partial \boldsymbol{\phi}} \sum_{i=1}^{I} \ell_i = \sum_{i=1}^{I} \frac{\partial \ell_i}{\partial \boldsymbol{\phi}}$$

- where these are given by (take 5 mins to compute it):

$$\ell_i = (\phi_0 + \phi_1 x_i - y_i)^2$$

$$\frac{\partial \ell_i}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial \ell_i}{\partial \phi_0} \\ \frac{\partial \ell_i}{\partial \phi_1} \end{bmatrix} = \begin{bmatrix} 2(\phi_0 + \phi_1 x_i - y_i) \\ 2x_i(\phi_0 + \phi_1 x_i - y_i) \end{bmatrix}$$

Gradient Descent: Linear Regression Example

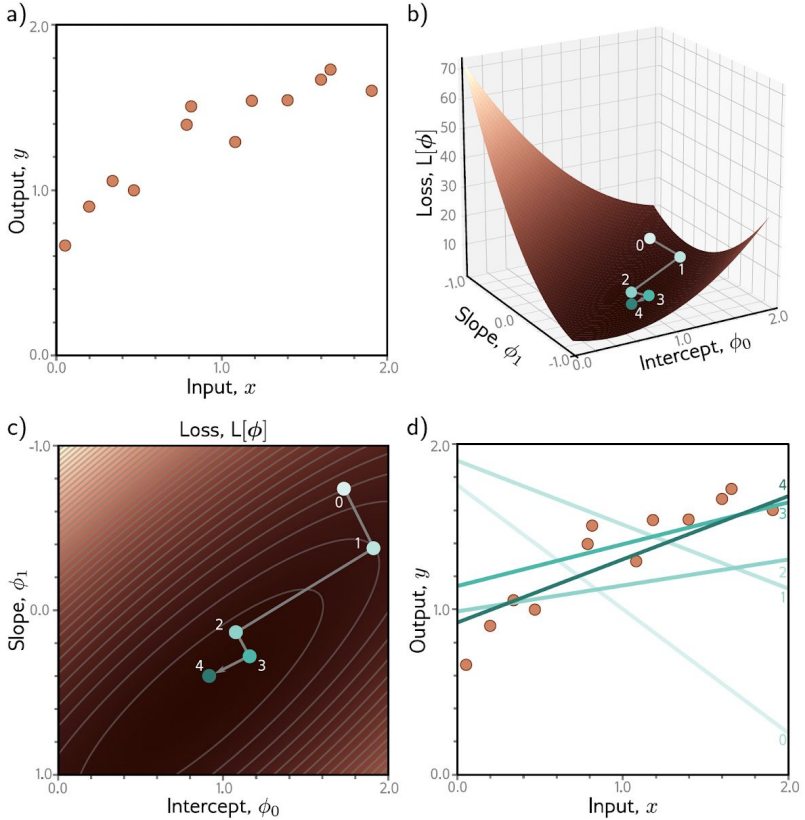


Figure 6.1 Gradient descent for the linear regression model. a) Training set of $I = 12$ input/output pairs $\{x_i, y_i\}$. b) Loss function showing iterations of gradient descent. We start at point 0 and move in the steepest downhill direction until we can improve no further to arrive at point 1. We then repeat this procedure. We measure the gradient at point 1 and move downhill to point 2 and so on. c) This can be visualized better as a heatmap, where the brightness represents the loss. After only four iterations, we are already close to the minimum. d) The model with the parameters at point 0 (lightest line) describes the data very badly, but each successive iteration improves the fit. The model with the parameters at point 4 (darkest line) is already a reasonable description of the training data.

Colab time: Gradient Descent

https://colab.research.google.com/drive/1kluwQQkqwcu9J025Sb754Ytwy2LXktV5

Is that it?!?!

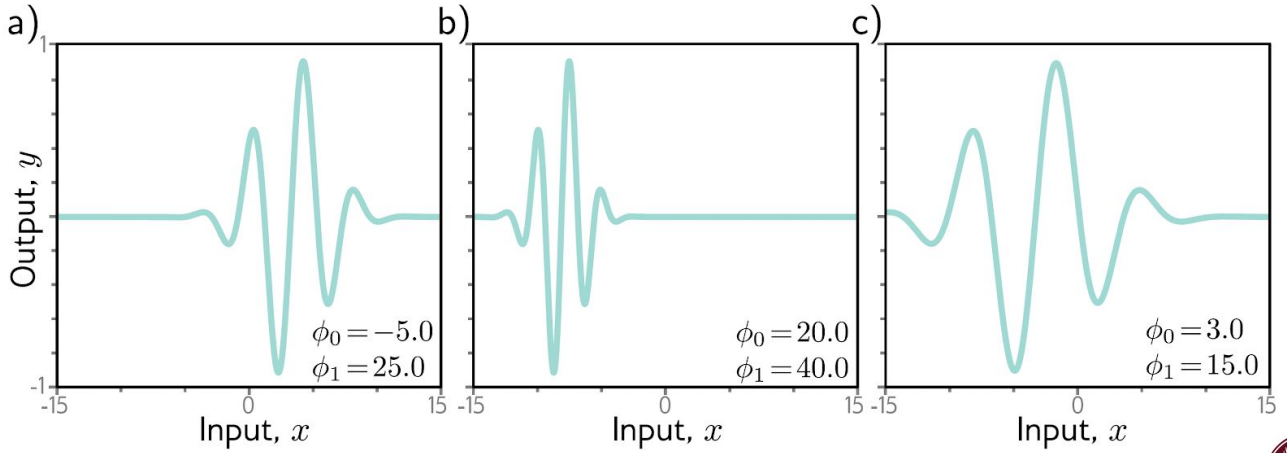
- For linear models, e.g. linear regression, the situation is usually good:
  - Convex functions always have a single optimum

Gabor Model

- Consider the following model (Gabor):

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{32.0}\right)$$

- This Gabor model maps scalar input x to scalar output y and consists of a sinusoidal component (creating an oscillatory function) multiplied by a negative exponential component (causing the amplitude to decrease as we move from the center).



Running Example for the Gabor Model

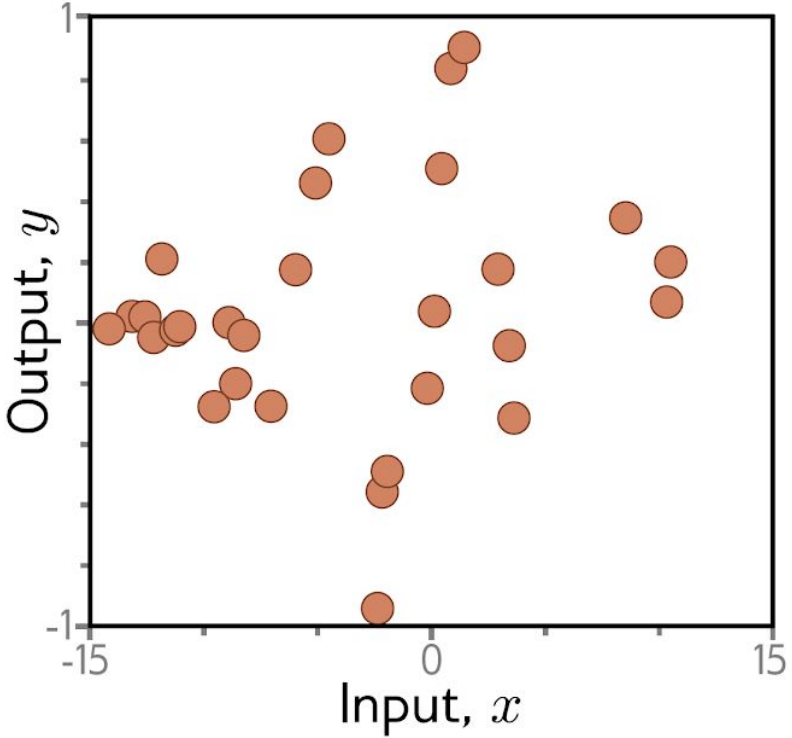


Figure 6.3 Training data for fitting the Gabor model. The training dataset contains 28 input/output examples $\{x_i, y_i\}$. These were created by uniformly sampling $x_i \in [-15, 15]$, passing the samples through a Gabor model with parameters $\boldsymbol{\phi} = [0.0, 16.6]^T$, and adding normally distributed noise.

Loss Landscape of MSE Applied to the Gabor Model

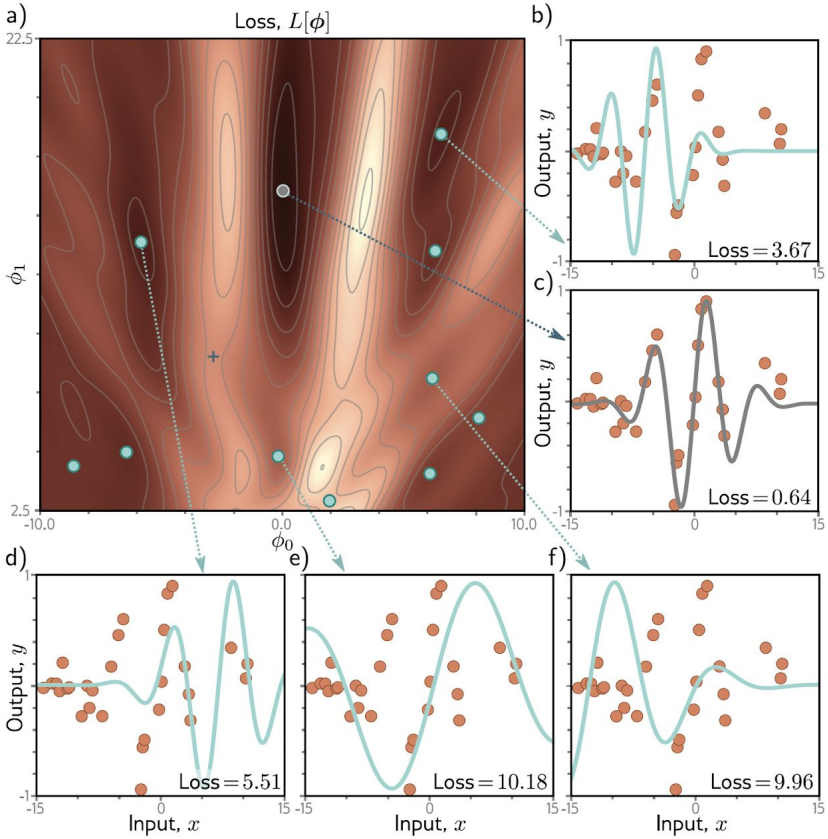
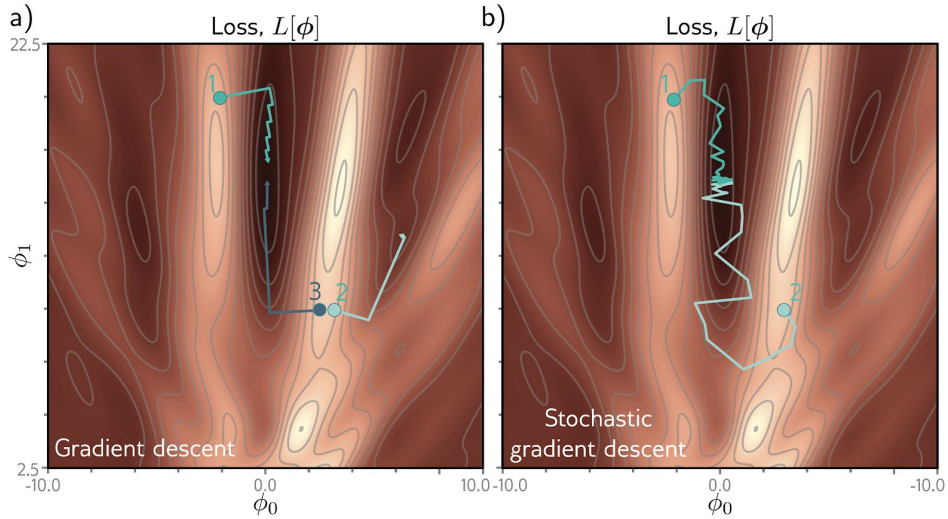


Figure 6.4 Loss function for the Gabor model. a) The loss function is non-convex, with multiple local minima (cyan circles) in addition to the global minimum (gray circle). It also contains saddle points where the gradient is locally zero, but the function increases in one direction and decreases in the other. The blue cross is an example of a saddle point; the function decreases as we move horizontally in either direction but increases as we move vertically. b–f) Models associated with the different minima. In each case, there is no small change that decreases the loss. Panel (c) shows the global minimum, which has a loss of 0.64.

Stochastic Gradient Descent

- Starts with initial parameters $\boldsymbol{\phi} = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates the same two steps of Gradient Descent, except…
- … The gradient is computed for each element (x_i, y_i) in the dataset: $\frac{\partial \ell_i}{\partial \boldsymbol{\phi}}$



SAPIENZA
UNIVERSITÀ DI ROMA

Batches and Epochs

- At each iteration, the algorithm chooses a random subset of the training data and computes the gradient from these examples alone.
  - This subset is known as a **minibatch** or **batch** for short.
- The update rule for the model parameters ϕ_t at iteration t is hence:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

Minibatch (or Batch)

- The batches are usually drawn from the dataset without replacement.
- The algorithm works through the training examples until it has used all the data, at which point it starts sampling from the full training dataset again.
- A *single pass through the entire training dataset is referred to as an epoch*.

Alternative Views of SGD

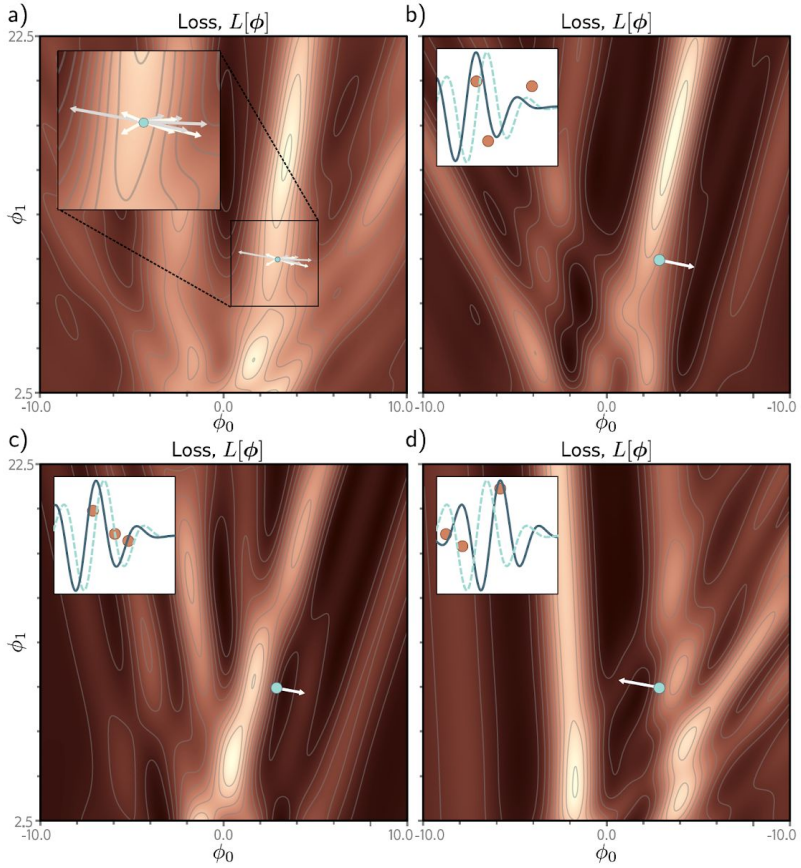


Figure 6.6 Alternative view of SGD for the Gabor model with a batch size of three. a) Loss function for the entire training dataset. At each iteration, there is a probability distribution of possible parameter changes (inset shows samples). These correspond to different choices of the three batch elements. b) Loss function for one possible batch. The SGD algorithm moves in the downhill direction on this function for a distance that is determined by the learning rate and the local gradient magnitude. The current model (dashed function in inset) changes to better fit the batch data (solid function). c) A different batch creates a different loss function and results in a different update. d) For this batch, the algorithm moves *downhill* with respect to the batch loss function but *uphill* with respect to the global loss function in panel (a). This is how SGD can escape local minima.

SGD: Pros

- Although it adds noise to the trajectory, it still improves the fit to a subset of the data at each iteration. Hence, the updates tend to be sensible even if they are not optimal.
- Given that it draws training examples without replacement and iterates through the dataset, the training examples all still contribute equally.
- It is less computationally expensive to compute the gradient from just a subset of the training data.
- It can (in principle) escape local minima.
- It reduces the chances of getting stuck near saddle points; it is likely that at least some of the possible batches will have a significant gradient at any point on the loss function.
- *There is some evidence that SGD finds parameters for neural networks that cause them to generalize well to new data in practice*.

SGD: Scheduled Learning Rate

- In practice, SGD is often applied with a learning rate schedule.
- The learning rate α starts at a high value and is decreased by a constant factor every N epochs.
  - The logic is that in the early stages of training, we want the algorithm to explore the parameter space, jumping from valley to valley to find a sensible region.
  - In later stages, we are roughly in the right place and are more concerned with fine-tuning the parameters, so we decrease α to make smaller changes.

SGD with Momentum

- A common modification to stochastic gradient descent is to add a momentum term.
- We update the parameters with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1},\end{aligned}$$

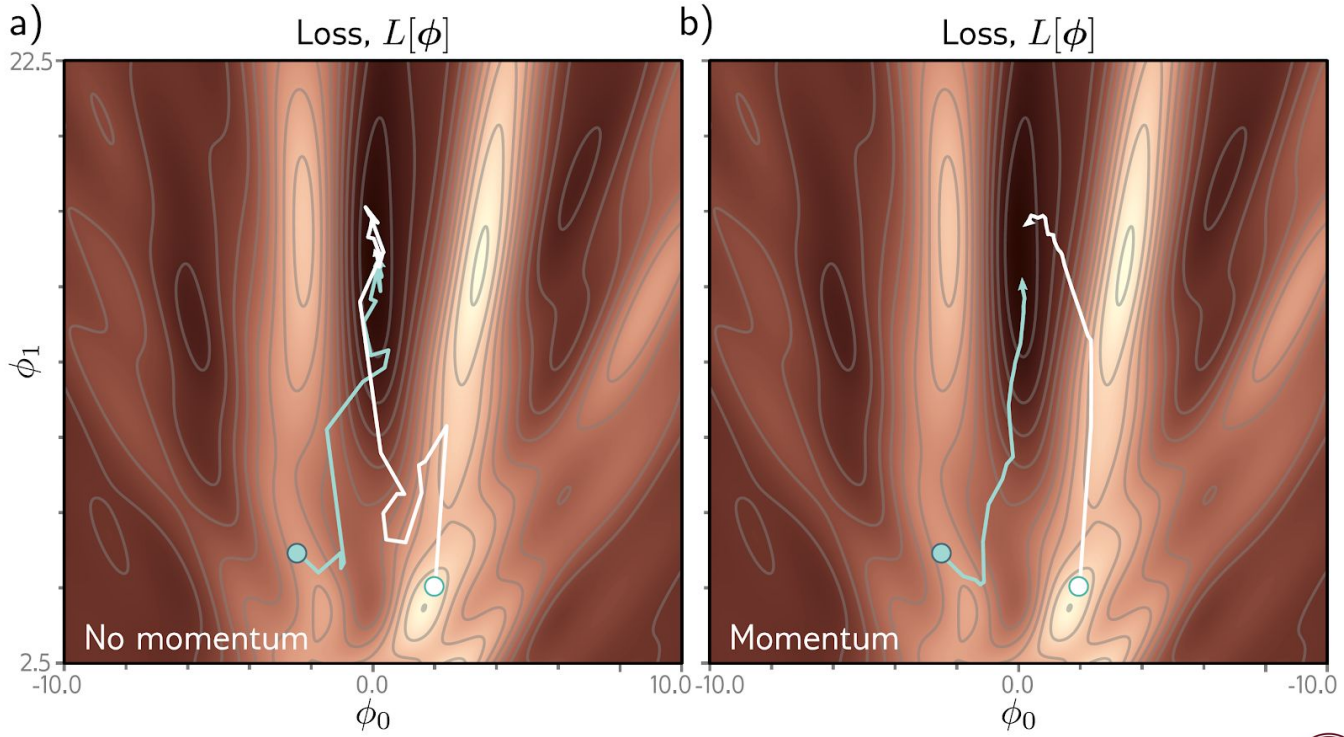
where $\mathbf{m}_t$ is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

SGD with Momentum

- The recursive formulation of the momentum calculation means that the gradient step is an infinite weighted sum of all the previous gradients, where the weights get smaller as we move back in time.
- The effective learning rate increases if all these gradients are aligned over multiple iterations but decreases if the gradient direction repeatedly changes as the terms in the sum cancel out.
- The overall effect is a smoother trajectory and reduced oscillatory behavior in valleys

SGD with Momentum

Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.



Colab time: Momentum

https://colab.research.google.com/drive/1RylYbFGwbpu8Wyb_kOI6pku5mZ4xFbhM

SGD with Nesterov Accelerated Momentum

- Nesterov accelerated momentum computes the gradients at this predicted point rather than at the current point:

$$(1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta\cdot\mathbf{m}_t + (1-\beta)\sum_{i\in\mathcal{B}_t}\frac{\partial \ell_i[\boldsymbol{\phi}_t-\alpha\cdot\mathbf{m}_t]}{\partial \boldsymbol{\phi}}\\ \boldsymbol{\phi}_{t+1} &\leftarrow \boldsymbol{\phi}_t-\alpha\cdot\mathbf{m}_{t+1},\end{aligned}$$

  where now the gradients are evaluated at $\phi_t - \alpha \cdot m_t$.

- One way to think about this is that the gradient term now corrects the path provided by momentum alone.

SGD with Nesterov Accelerated Momentum

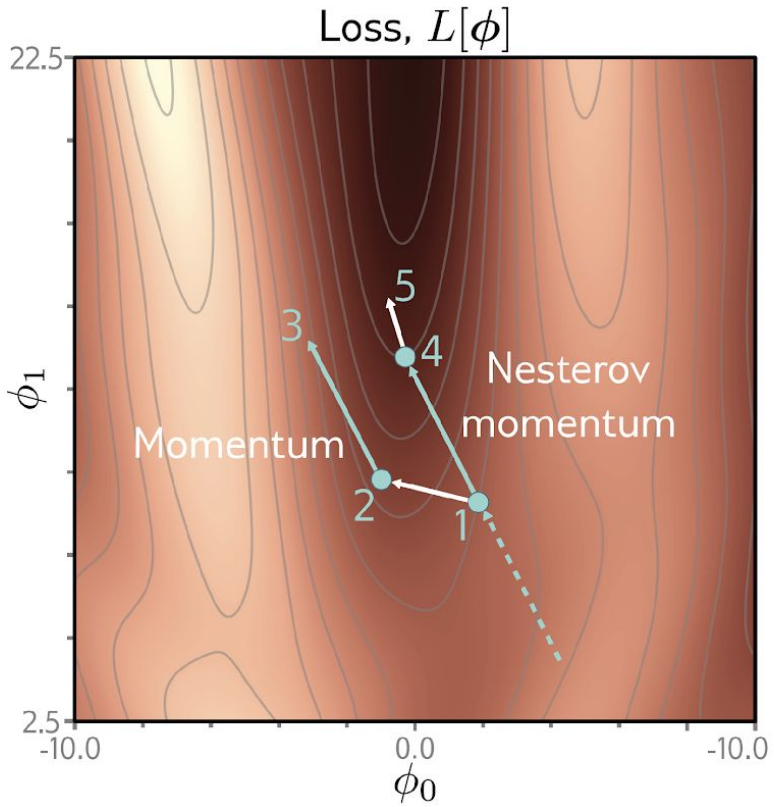


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

SGD with Adam

- Gradient descent with a fixed step size has the following undesirable property: it makes large adjustments to parameters associated with large gradients (where perhaps we should be more cautious) and small adjustments to parameters associated with small gradients (where perhaps we should explore further).
- When the gradient of the loss surface is much steeper in one direction than another, it is difficult to choose a learning rate that (i) makes good progress in both directions and (ii) is stable

SGD with Adam

- A straightforward approach is to normalize the gradients so that we move a fixed distance (governed by the learning rate) in each direction. To do this, we first measure the gradient m_{t+1} and the pointwise squared gradient v_{t+1}:

$$\begin{aligned} \mathbf{m}_{t+1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2 \end{aligned}$$

- Then we apply the update rule:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\mathbf{m}_{t+1}}{\sqrt{\mathbf{v}_{t+1}} + \epsilon}$$

SGD with Adam

- Adam extends this idea by refining m_{t+1} and v_{t+1} calculation

$$\begin{aligned}\mathbf{m}_{t+1} &\leftarrow \beta \cdot \mathbf{m}_t + (1-\beta)\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t+1} &\leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma)\left(\frac{\partial L[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}\right)^2\end{aligned}$$

  where β and γ are the momentum coefficients for the two statistics.
- Using momentum is equivalent to taking a weighted average over the history of each of these statistics. At the start of the procedure, all the previous measurements are effectively zero, resulting in unrealistically small estimates. Consequently, we modify these statistics using the rule:

$$\begin{aligned}\tilde{\mathbf{m}}_{t+1} &\leftarrow \frac{\mathbf{m}_{t+1}}{1-\beta^{t+1}} \\ \tilde{\mathbf{v}}_{t+1} &\leftarrow \frac{\mathbf{v}_{t+1}}{1-\gamma^{t+1}}.\end{aligned}$$

SGD with Adam

- Finally, we update the parameters as before, but with the modified terms:

$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \frac{\tilde{\mathbf{m}}_{t+1}}{\sqrt{\tilde{\mathbf{v}}_{t+1}} + \epsilon}.$$

- In stochastic settings

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

$$\mathbf{v}_{t+1} \leftarrow \gamma \cdot \mathbf{v}_t + (1-\gamma) \sum_{i \in \mathcal{B}_t} \left(\frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}} \right)^2$$

SGD with Adam

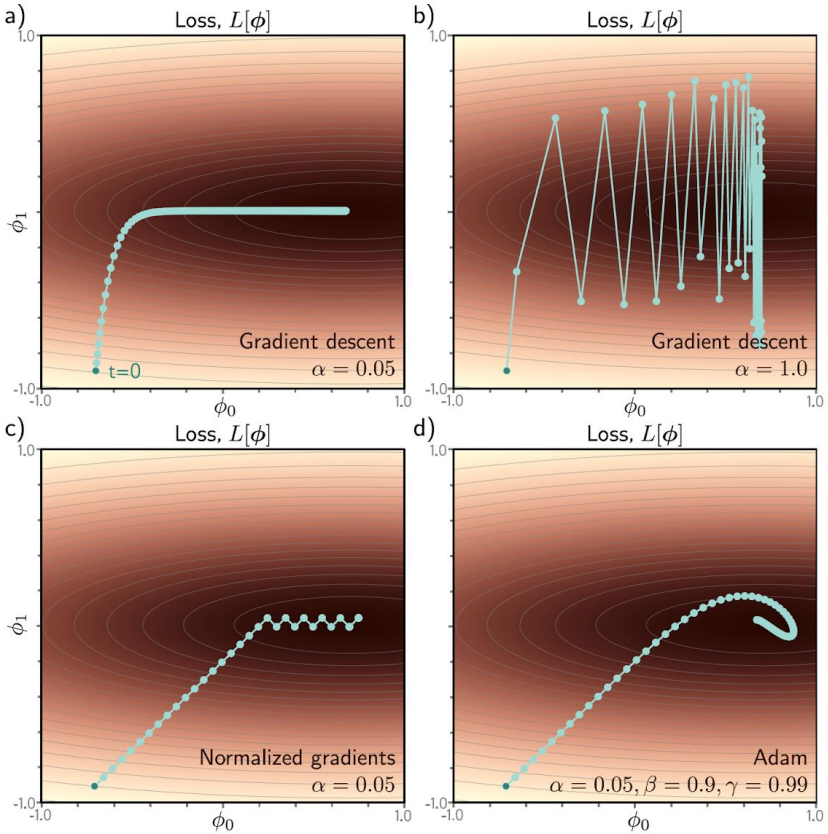


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

Colab time: Adam

https://colab.research.google.com/drive/1rIvw4qDWGTq2EOpVjgOcNkobW-U6Gng9

Hyperparameters of the Training Algorithms

- The choices of learning algorithm, batch size, learning rate schedule, and momentum coefficients are all considered hyperparameters of the training algorithm
    - these directly affect the final model performance but are distinct from the model parameters.
- Choosing these can be more art than science, and it's common to train many models with different hyperparameters and choose the best one.
- This is known as *hyperparameter search*.

Fitting Models: Backprop

(Chapter 6)

SAPIENZA
UNIVERSITÀ DI ROMA

Setting Example

- Consider a network f[**x**, **ϕ**] with multivariate input **x**, parameters **ϕ**, and three hidden layers $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\ \mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\ \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3,\end{aligned}$$

- We also have individual loss terms ℓ_i

$$L[\boldsymbol{\phi}] = \sum_{i=1}^{I} \ell_i$$

Setting Example

- Let's assume we use some sort of Stochastic Gradient Descent

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

- Therefore we need to compute the derivatives:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} \quad \text{and} \quad \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k}$$

- This is done through Backpropapagation, a.k.a. Backprop

Backprop: Forward Pass

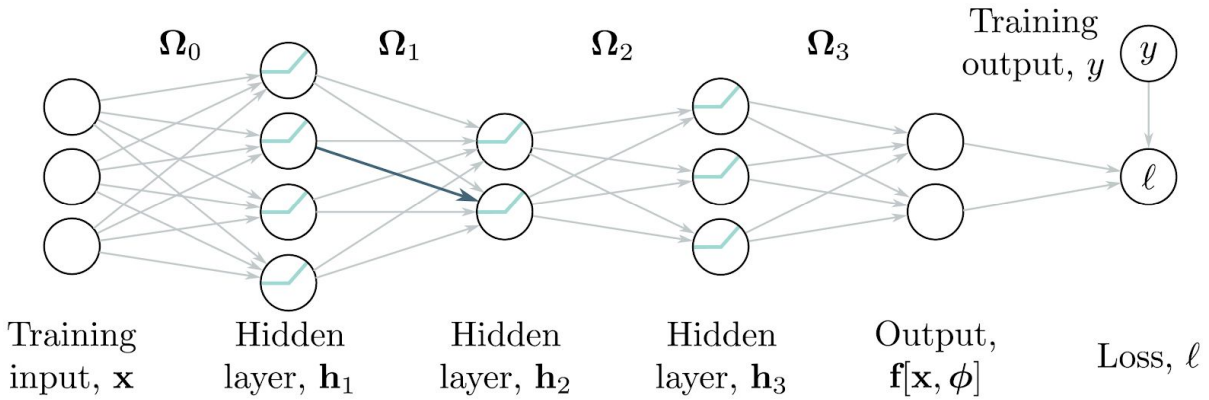


Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss ℓ with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the blue weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the blue weight will double too. Hence, to compute the derivatives of the weights, we need to calculate and store the activations at the hidden layers. This is known as the *forward pass* since it involves running the network equations sequentially.

Backprop: Backward Pass

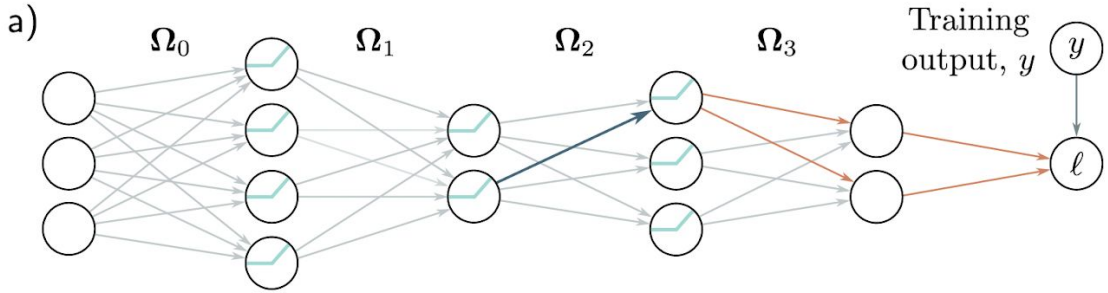


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

Backprop: Backward Pass

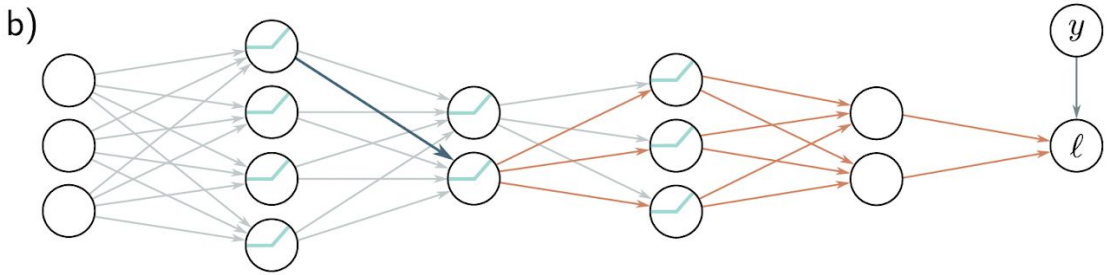


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

Backprop: Backward Pass

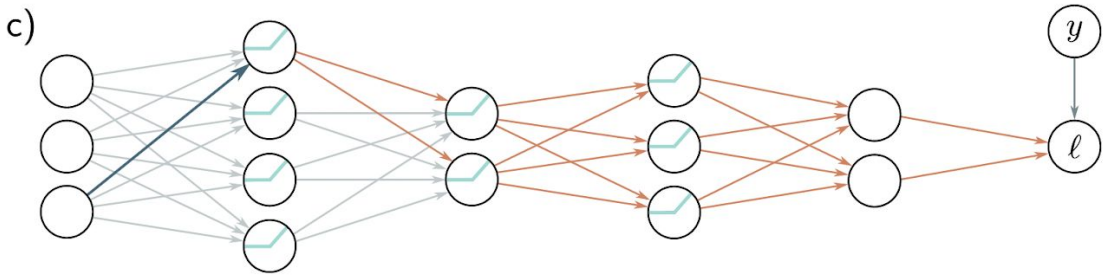


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

Overall Backprop Algorithm: Forward Pass

- Consider a deep neural network $f[\mathbf{x}_i, \boldsymbol{\phi}]$ that takes input $\mathbf{x}_i$, has K hidden layers with ReLU activations, and individual loss term $\ell i = l[f[\mathbf{x}_i, \boldsymbol{\phi}], \mathbf{y}_i]$.
- The goal of backpropagation is to compute the derivatives $\partial \ell_i / \partial \boldsymbol{\beta}_k$ and $\partial \ell_i / \partial \boldsymbol{\Omega}_k$ with respect to the biases $\boldsymbol{\beta}_k$ and weights $\boldsymbol{\Omega}_k$.

Forward pass: We compute and store the following quantities:

$$
\begin{aligned}
\mathbf{f}_0 &= \boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}[\mathbf{f}_{k-1}] && k \in \{1, 2, \ldots, K\} \\
\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k. && k \in \{1, 2, \ldots, K\}
\end{aligned}
$$

Overall Backprop Algorithm: Backward Pass

Backward pass: We start with the derivative $\partial \ell_i / \partial \mathbf{f}_K$ of the loss function ℓ_i with respect to the network output $\mathbf{f}_K$ and work backward through the network:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} && k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial \ell_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T && k \in \{K, K-1, \ldots, 1\} \\
\frac{\partial \ell_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial \ell_i}{\partial \mathbf{f}_k} \right), && k \in \{K, K-1, \ldots, 1\}
\end{aligned}$$

where $\odot$ denotes pointwise multiplication, and $\mathbb{I}[\mathbf{f}_{k-1} > 0]$ is a vector containing ones where $\mathbf{f}_{k-1}$ is greater than zero and zeros elsewhere. Finally, we compute the derivatives with respect to the first set of biases and weights:

$$\begin{aligned}
\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \\
\frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_0} &= \frac{\partial \ell_i}{\partial \mathbf{f}_0} \mathbf{x}_i^T.
\end{aligned}$$

SAPIENZA
UNIVERSITÀ DI ROMA

Colab time: Backpropagation

https://colab.research.google.com/drive/1L_MejkotJy4quD9DtYaLviBZN3Rm33Tx

Backprop: A Toy Example

- Consider a model f[**x**,**ϕ**] with eight scalar parameters

  ϕ = {β_0, ω_0, β_1, ω_1, β_2, ω_2, β_3, ω_3} that consists of a composition of the functions sin[•], exp[•], and cos[•]:

$$\mathrm{f}[x, \boldsymbol{\phi}] = \beta_3 + \omega_3 \cdot \cos\Big[\beta_2 + \omega_2 \cdot \exp\big[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x]\big]\Big]$$

  and a least squares loss function $\mathrm{L}[\boldsymbol{\phi}] = \sum_i \ell_i$ with individual terms:

$$\ell_i = (\mathrm{f}[x_i, \boldsymbol{\phi}] - y_i)^2$$

A Toy Example: What Do We Need to Do?

- We need to compute

$$\frac{\partial \ell_i}{\partial \beta_0}, \quad \frac{\partial \ell_i}{\partial \omega_0}, \quad \frac{\partial \ell_i}{\partial \beta_1}, \quad \frac{\partial \ell_i}{\partial \omega_1}, \quad \frac{\partial \ell_i}{\partial \beta_2}, \quad \frac{\partial \ell_i}{\partial \omega_2}, \quad \frac{\partial \ell_i}{\partial \beta_3}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial \omega_3}$$

- Can we do it manually?
- Sure… But…

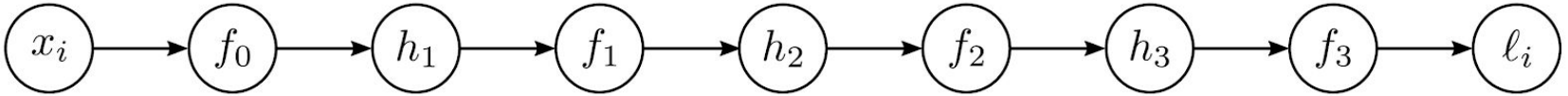
A Toy Example: What Do We Need to Do?

- Look at this guy…

$$\frac{\partial \ell_i}{\partial \omega_0} = -2\left(\beta_3 + \omega_3 \cdot \cos\left[\beta_2 + \omega_2 \cdot \exp\left[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\right]\right] - y_i\right)$$
$$\cdot \omega_1\omega_2\omega_3 \cdot x_i \cdot \cos[\beta_0 + \omega_0 \cdot x_i] \cdot \exp\left[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\right]$$
$$\cdot \sin\left[\beta_2 + \omega_2 \cdot \exp\left[\beta_1 + \omega_1 \cdot \sin[\beta_0 + \omega_0 \cdot x_i]\right]\right].$$

- Lots of redundancy

A Toy Example: The Forward Pass

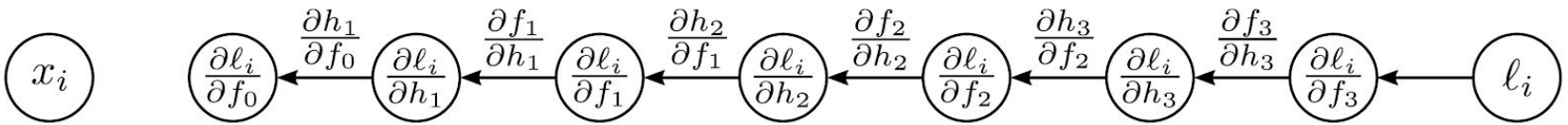


Forward pass: We treat the computation of the loss as a series of calculations:

$$
\begin{aligned}
f_0 &= \beta_0 + \omega_0 \cdot x_i \\
h_1 &= \sin[f_0] \\
f_1 &= \beta_1 + \omega_1 \cdot h_1 \\
h_2 &= \exp[f_1] \\
f_2 &= \beta_2 + \omega_2 \cdot h_2 \\
h_3 &= \cos[f_2] \\
f_3 &= \beta_3 + \omega_3 \cdot h_3 \\
\ell_i &= (f_3 - y_i)^2.
\end{aligned}
$$

We compute and store the values of the intermediate variables f_k and h_k

A Toy Example: The Backward Pass



Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

$$\frac{\partial \ell_i}{\partial f_3}, \quad \frac{\partial \ell_i}{\partial h_3}, \quad \frac{\partial \ell_i}{\partial f_2}, \quad \frac{\partial \ell_i}{\partial h_2}, \quad \frac{\partial \ell_i}{\partial f_1}, \quad \frac{\partial \ell_i}{\partial h_1}, \quad \text{and} \quad \frac{\partial \ell_i}{\partial f_0}.$$

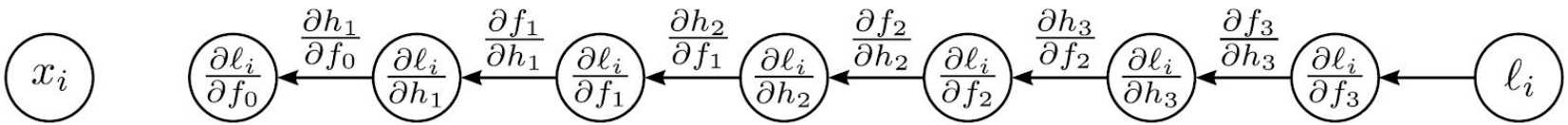
The first of these derivatives is straightforward:

$$\frac{\partial \ell_i}{\partial f_3} = 2(f_3 - y_i).$$

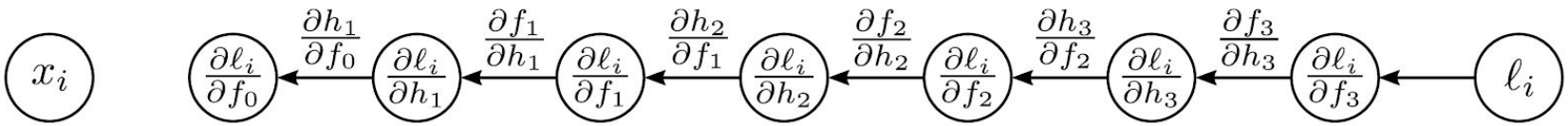
The next derivative can be calculated using the chain rule:

$$\frac{\partial \ell_i}{\partial h_3} = \frac{\partial f_3}{\partial h_3} \frac{\partial \ell_i}{\partial f_3}.$$

A Toy Example: The Backward Pass



A Toy Example: The Backward Pass

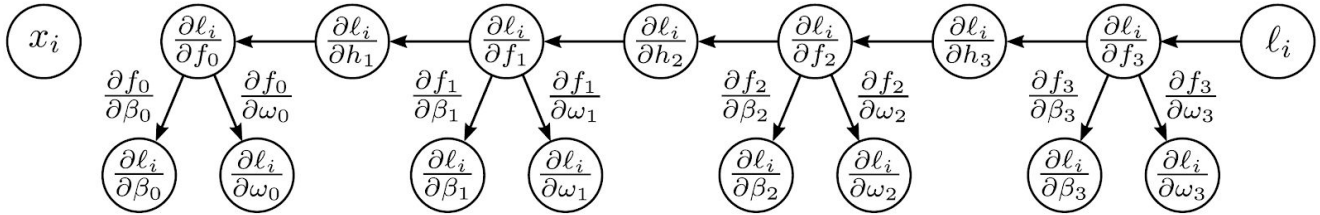


We continue in this way, computing the derivatives of the output with respect to these intermediate quantities (figure 7.4):

$$\begin{aligned}
\frac{\partial \ell_i}{\partial f_2} &= \frac{\partial h_3}{\partial f_2}\left(\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_2} &= \frac{\partial f_2}{\partial h_2}\left(\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_1} &= \frac{\partial h_2}{\partial f_1}\left(\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial h_1} &= \frac{\partial f_1}{\partial h_1}\left(\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right) \\
\frac{\partial \ell_i}{\partial f_0} &= \frac{\partial h_1}{\partial f_0}\left(\frac{\partial f_1}{\partial h_1}\frac{\partial h_2}{\partial f_1}\frac{\partial f_2}{\partial h_2}\frac{\partial h_3}{\partial f_2}\frac{\partial f_3}{\partial h_3}\frac{\partial \ell_i}{\partial f_3}\right).
\end{aligned} \tag{7.12}$$

In each case, we have already computed the quantities in the brackets in the previous step, and the last term has a simple expression. These equations embody Observation 2 from the previous section (figure 7.2); we can reuse the previously computed derivatives if we calculate them in reverse order.

A Toy Example: The Backward Pass



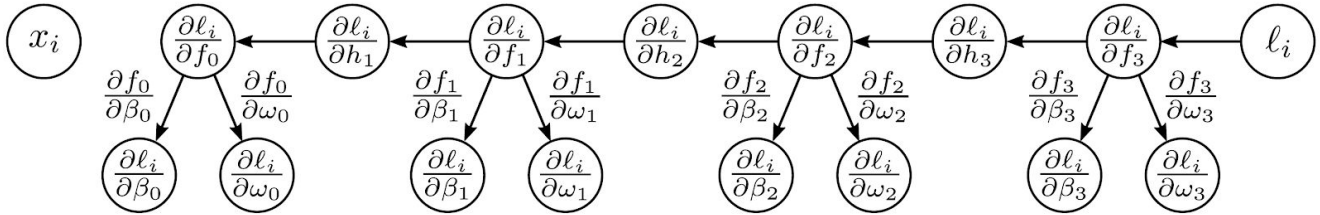
Backward pass #2: Finally, we consider how the loss ℓ_i changes when we change the parameters $\beta_\bullet$ and $\omega_\bullet$. Once more, we apply the chain rule (figure 7.5):

$$\begin{aligned}\frac{\partial \ell_i}{\partial \beta_k} &= \frac{\partial f_k}{\partial \beta_k}\frac{\partial \ell_i}{\partial f_k} \\ \frac{\partial \ell_i}{\partial \omega_k} &= \frac{\partial f_k}{\partial \omega_k}\frac{\partial \ell_i}{\partial f_k}.\end{aligned}$$

In each case, the second term on the right-hand side was computed in equation When $k > 0$, we have $f_k = \beta_k + \omega_k \cdot h_k$, so:

$$\frac{\partial f_k}{\partial \beta_k} = 1 \quad \text{and} \quad \frac{\partial f_k}{\partial \omega_k} = h_k.$$

A Toy Example: The Backward Pass



This is consistent with Observation 1 from the previous section; the effect of a change in the weight ω_k is proportional to the value of the source variable h_k (which was stored in the forward pass). The final derivatives from the term $f_0 = \beta_0 + \omega \cdot x_i$ are:

$$\frac{\partial f_0}{\partial \beta_0} = 1 \qquad \text{and} \qquad \frac{\partial f_0}{\partial \omega_0} = x_i.$$

Colab time: Backpropagation

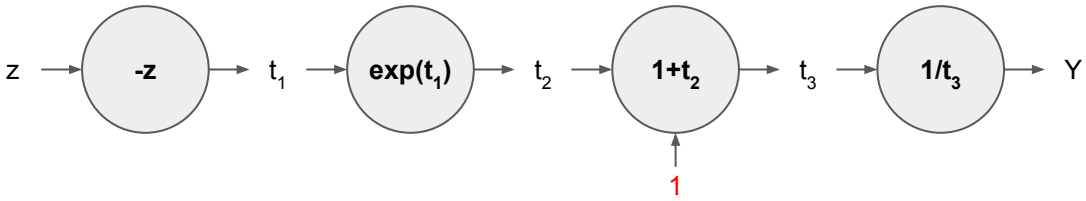
https://colab.research.google.com/drive/1-3F50CNG14gLB6fVCc11B16-Mi_jG6DP

Computation Graph

Computation Graph: Logistic Regression

```
def logistic(z):
  return 1. / (1. + np.exp(z))
```

Z = 1.5
Y = logistic(1.5)



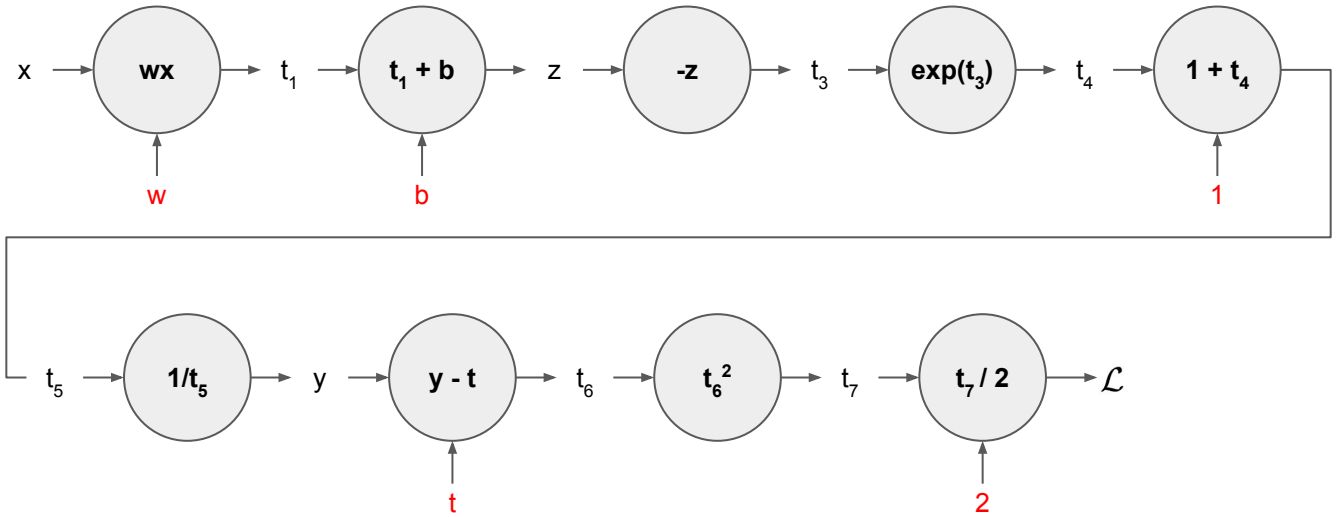
Computation Graph: Include the Loss

Original program:

$$z = wx + b$$

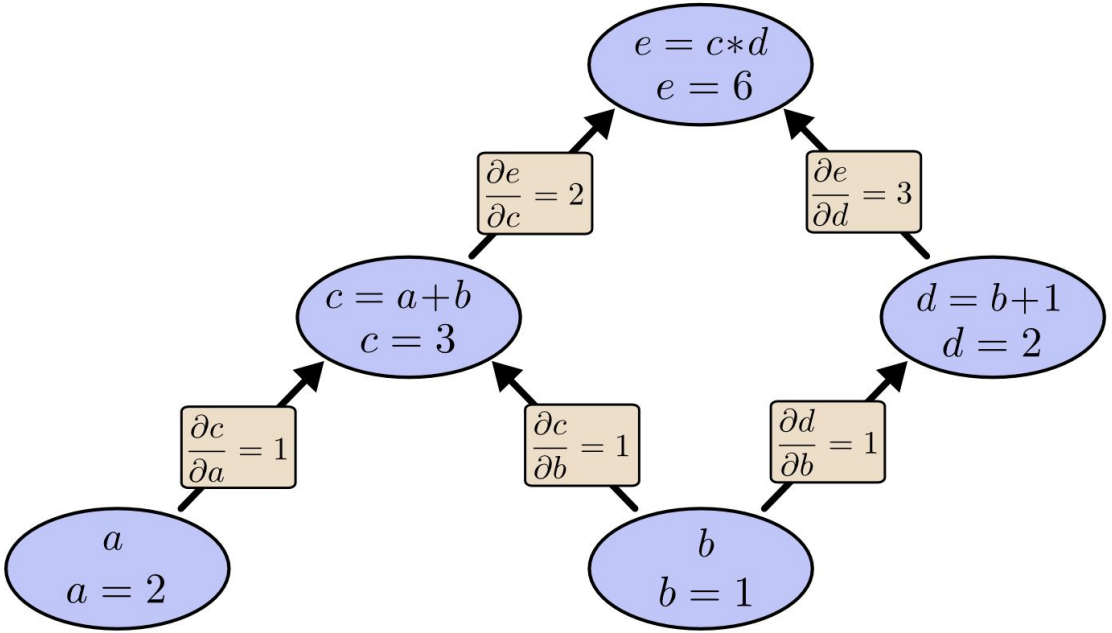
$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

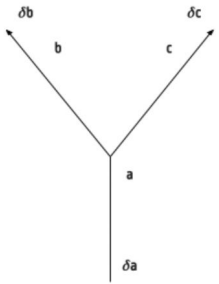
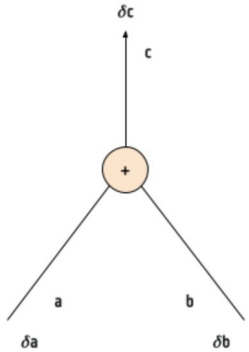
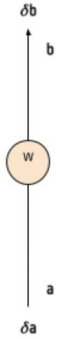
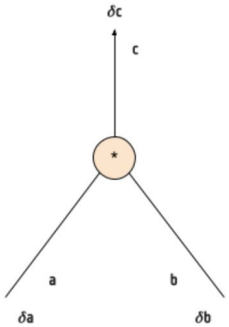


Backprop on the Computation Graph: Reverse Mode

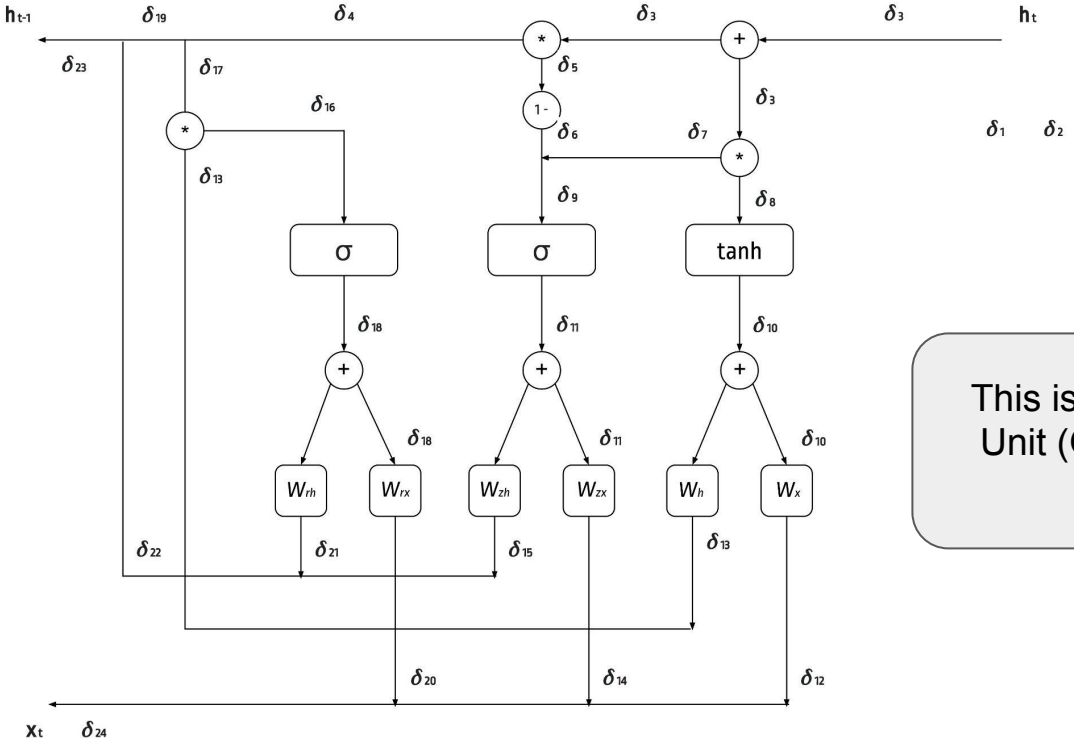
- (Reverse Mode) Autodiff starts at an output of the graph and moves towards the beginning.
- At each node, it merges all paths which originated at that node.
- Example: (a+b)(b+1)



Backprop on the Computation Graph: Reverse Mode

SPLIT	ADDITION	FUNCTION	MATRIX MULTIPLY	HADAMARD PRODUCT
				
$c = a; b = a$	$c = a + b$	$b = f(a)$	$b = Wa$	$c = a * b$
$\delta_a = \delta_b + \delta_c$	$\delta_a = \delta_c; \delta_b = \delta_c$	$\delta_a = \delta_b * f'(a)$	$\delta_a = \delta_b * W^T$	$\delta_a = \delta_c * b$
				$\delta_b = \delta_c * a$

Things can get complicated :)



This is a Gated Recurrent Unit (GRU) Computation Graph

Parameter Initialization

Why Do We Care?

- Consider that during the forward pass, each set of pre-activations $\mathbf{f}_k$ is computed as:

$$\begin{aligned}\mathbf{f}_k &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k \\ &= \boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathrm{a}[\mathbf{f}_{k-1}],\end{aligned}$$

  where a[•] applies the ReLU functions and $\boldsymbol{\Omega}_k$ and $\boldsymbol{\beta}_k$ are the weights and biases.
- Imagine that we initialize all the biases to zero and the elements of $\boldsymbol{\Omega}_k$ according to a normal distribution with mean zero and variance σ^2.

Low Variance: small σ^2 (e.g., 10^{-5})

- Each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of $\mathbf{h}_k$ where the weights are very small; the result will likely have a smaller magnitude than the input.
  - In addition, the ReLU function clips values less than zero, so the range of $\mathbf{h}_k$ will be half that of $\mathbf{f}_{k-1}$.
- Consequently, the magnitudes of the pre-activations at the hidden layers will get smaller and smaller as we progress through the network.

High Variance: large σ^2 (e.g., 10^5)

- Each element of $\boldsymbol{\beta}_k + \boldsymbol{\Omega}_k \mathbf{h}_k$ will be a weighted sum of h_k where the weights are very large
- The result is likely to have a much larger magnitude than the input.
  - The ReLU function halves the range of the inputs, but if σ^2 is large enough, the magnitudes of the pre-activations will still get larger as we progress through the network.

Vanishing and Exploding Gradients

- If the values of **Ω** are not initialized sensibly, then the gradient magnitudes may decrease or increase uncontrollably during the backward pass.
    - These cases are known as the **vanishing gradient** problem and the **exploding gradient** problem, respectively.
- In the former case, updates to the model become vanishingly small. In the latter case, they become unstable.

Initialization for Forward Pass

- Consider the computation between adjacent pre-activations **f** and **f′** with dimensions D_h and $D_{h'}$, respectively:

$$\begin{aligned} \mathbf{h} &= \mathbf{a}[\mathbf{f}], \\ \mathbf{f}' &= \boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{h} \end{aligned}$$

- Assume the pre-activations $\mathbf{f}_j$ in the input layer **f** have variance $\sigma^2_{\mathbf{f}}$.
- Consider initializing the biases $\boldsymbol{\beta}_i$ to zero and the weights $\boldsymbol{\Omega}_{ij}$ as normally distributed with mean zero and variance $\sigma^2_{\boldsymbol{\Omega}}$.

Expectation (mean) $\mathbf{E}[\mathbf{f}_i]$

$$
\begin{aligned}
\mathbb{E}[f_i'] &= \mathbb{E}\left[\beta_i + \sum_{j=1}^{D_h} \Omega_{ij} h_j\right] \\
&= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h} \mathbb{E}\left[\Omega_{ij} h_j\right] \\
&= \mathbb{E}\left[\beta_i\right] + \sum_{j=1}^{D_h} \mathbb{E}\left[\Omega_{ij}\right] \mathbb{E}\left[h_j\right] \\
&= 0 + \sum_{j=1}^{D_h} 0 \cdot \mathbb{E}\left[h_j\right] = 0,
\end{aligned}
$$

Variance of $\mathbf{f}_i$

$$
\begin{aligned}
\sigma^2_{f'} &= \mathbb{E}[f_i'^2] - \mathbb{E}[f_i']^2 \\
&= \mathbb{E}\left[\left(\beta_i + \sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right] - 0 \\
&= \mathbb{E}\left[\left(\sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right] \\
&= \sum_{j=1}^{D_h} \mathbb{E}\left[\Omega_{ij}^2\right] \mathbb{E}\left[h_j^2\right] \\
&= \sum_{j=1}^{D_h} \sigma_\Omega^2 \mathbb{E}\left[h_j^2\right] = \sigma_\Omega^2 \sum_{j=1}^{D_h} \mathbb{E}\left[h_j^2\right]
\end{aligned}
$$

Variance of $\mathbf{f}_i$

$$
\begin{aligned}
\sigma_{f'}^2 &= \mathbb{E}[f_i'^2] - \mathbb{E}[f_i']^2 \\
&= \mathbb{E}\left[\left(\beta_i + \sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right] - 0 \\
&= \mathbb{E}\left[\left(\sum_{j=1}^{D_h} \Omega_{ij} h_j\right)^2\right] \\
&= \sum_{j=1}^{D_h} \mathbb{E}\left[\Omega_{ij}^2\right] \mathbb{E}\left[h_j^2\right] \\
&= \sum_{j=1}^{D_h} \sigma_\Omega^2 \mathbb{E}\left[h_j^2\right] = \sigma_\Omega^2 \sum_{j=1}^{D_h} \mathbb{E}\left[h_j^2\right]
\end{aligned}
$$

- Assuming that the input distribution of pre-activations $\mathbf{f}_j$ is symmetric about zero, half of these pre-activations will be clipped by the ReLU function, and the second moment $E[h_j]^2$ will be half the variance σ^2_f of $\mathbf{f}_j$

$$\sigma_{f'}^2 = \sigma_\Omega^2 \sum_{j=1}^{D_h} \frac{\sigma_f^2}{2} = \frac{1}{2} D_h \sigma_\Omega^2 \sigma_f^2$$

Variance of $\mathbf{f}_i$

$$\sigma^2_{f'} = \sigma^2_\Omega \sum_{j=1}^{D_h} \frac{\sigma^2_f}{2} = \frac{1}{2} D_h \sigma^2_\Omega \sigma^2_f$$

- What if we want the variance $\sigma^2_{\mathbf{f}'}$ of the subsequent pre-activations $\mathbf{f}'$ to be the same as the variance $\sigma^2_{\mathbf{f}}$ of the original pre-activations f during the forward pass?
- We should set:

$$\sigma^2_\Omega = \frac{2}{D_h}$$

  where D_h is the dimension of the original layer to which the weights were applied.
- This is known as **He initialization**.

Initialization for Backward Pass

- A similar argument establishes how the variance of the gradients ∂l/∂fk changes during the backward pass.

$$\sigma_{\Omega}^{2} = \frac{2}{D_{h'}}$$

where $D_{h'}$ is the dimension of the layer that the weights feed into.

Initialization for both Forward and Backward Pass

- If the weight matrix **Ω** is not square (i.e., there are different numbers of hidden units in the two adjacent layers, so D_h and $D_{h'}$ differ), then it is not possible to choose the variance to satisfy both equations simultaneously

$$\sigma_\Omega^2 = \frac{2}{D_h} \qquad \sigma_\Omega^2 = \frac{2}{D_{h'}}$$

- One possible compromise is to use the mean (D_h + $D_{h'}$)/2 as a proxy for the number of terms, which gives:

$$\sigma_\Omega^2 = \frac{4}{D_h + D_{h'}}$$

Initialization for both Forward and Backward Pass

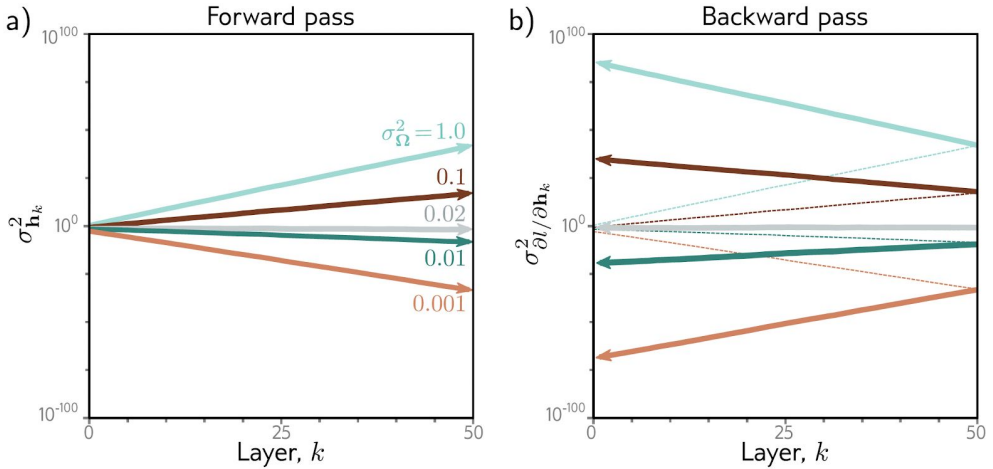


Figure 7.7 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100-dimensional input $\mathbf{x}$ initialized from a standard normal distribution, a single fixed target $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero, and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_{\Omega} \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_{\Omega} = 2/D_h = 0.02$), the variance is stable. However, for larger values, it increases rapidly, and for smaller values, it decreases rapidly (note log scale). b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are known as the *exploding gradient* and *vanishing gradient* problems, respectively.

It is a mess… innit?


```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
    nn.Linear(D_i, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
    if isinstance(layer_in, nn.Linear):
        nn.init.kaiming_uniform(layer_in.weight)
        layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 random data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

# loop over the dataset 100 times
for epoch in range(100):
    epoch_loss = 0.0
    # loop over batches
    for i, data in enumerate(data_loader):
        # retrieve inputs and labels for this batch
        x_batch, y_batch = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward pass
        pred = model(x_batch)
        loss = criterion(pred, y_batch)
        # backward pass
        loss.backward()
        # SGD update
        optimizer.step()
        # update statistics
        epoch_loss += loss.item()
    # print error
    print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
    # tell scheduler to consider updating learning rate
    scheduler.step()
```

Deep Learning

End of Lecture

04 - Model Training. Fitting & Backprop

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri