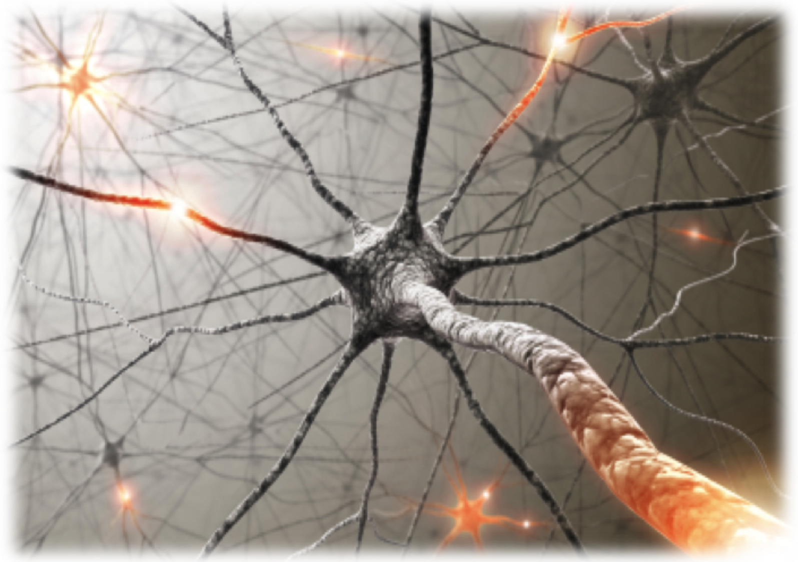
Machine learning: backpropagation



- In this module, I'll discuss **backpropagation**, an algorithm to automatically compute gradients.
- It is generally associated with training neural networks, but actually it is much more general and applies to any function.

Motivation: regression with four-layer neural networks

Loss on one example:

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$

Stochastic gradient descent:

$$\mathbf{V}_1 \leftarrow \mathbf{V}_1 - \eta \nabla_{\mathbf{V}_1} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_2 \leftarrow \mathbf{V}_2 - \eta \nabla_{\mathbf{V}_2} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_3 \leftarrow \mathbf{V}_3 - \eta \nabla_{\mathbf{V}_3} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

How to get the gradient without doing manual work?

- So far, we've defined neural networks, which take an initial feature vector $\phi(x)$ and sends it through a sequence of matrix multiplications and non-linear activations σ. At the end, we take the dot product between a weight vector $\mathbf{w}$ to produce the score.
- In regression, we predict the score, and use the squared loss, which looks at the squared difference betwen the score and the target y.
- Recall that we can use stochastic gradient descent to optimize the training loss (which is an average over the per-example losses). Now, we need to update all the weight matrices, not just a single weight vector. This can be done by taking the gradient with respect to each weight vector/matrix separately, and updating the respective weight vector/matrix by subtracting the gradient times a step size η.
- We can now proceed to take the gradient of the loss function with respect to the various weight vector/matrices. You should know how to do this: just apply the chain rule. But grinding through this complex expression by hand can be quite tedious. If only we had a way for this to be done automatically for us...

Computation graphs

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3\sigma(\mathbf{V}_2\sigma(\mathbf{V}_1\phi(x)))) - y)^2$$

Definition: computation graph

A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions.

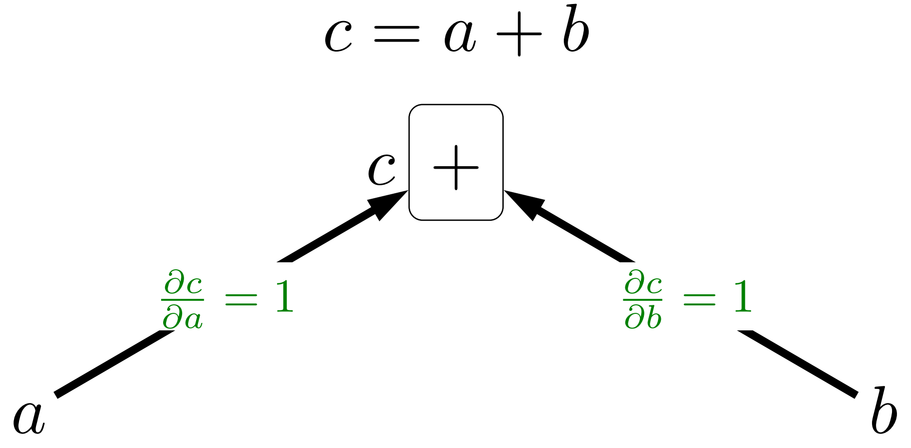
Upshot: compute gradients via general **backpropagation** algorithm

Purposes:

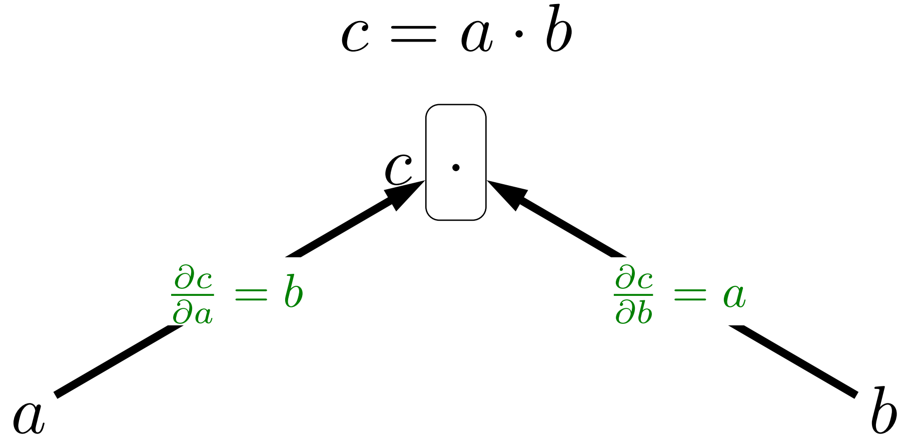
- Automatically compute gradients (how TensorFlow and PyTorch work)
- Gain insight into modular structure of gradient computations

- Enter computation graphs, which will rescue us.
- A computation graph is a directed acyclic graph that represents an arbitrary mathematical expression. The root of that node represents the final expression, and the other nodes represent intermediate subexpressions.
- After having constructed the graph, we can compute all the gradients we want by running the general-purpose backpropagation algorithm, which operates on an arbitrary computation graph.
- There are two purposes to using computation graphs. The first and most obvious one is that it avoids having us to do pages of calculus, and instead delegates this to a computer. This is what packages such as TensorFlow or PyTorch do, and essentially all non-trivial deep learning models are trained like this.
- The second purpose is that by defining the graph, we can gain more insight into the nature of how gradients are computed in a modular way.

Functions as boxes



$c = a + b$
c
$+$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
a
b



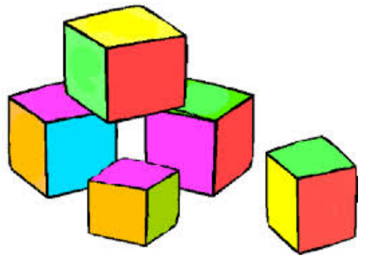
$c = a \cdot b$
c
$\cdot$
$\frac{\partial c}{\partial a} = b$
$\frac{\partial c}{\partial b} = a$
a
b

$$(a + \epsilon) + b = c + 1\epsilon$$
$$a + (b + \epsilon) = c + 1\epsilon$$

$$(a + \epsilon)b = c + b\epsilon$$
$$a(b + \epsilon) = c + a\epsilon$$

Gradients: how much does c change if a or b changes?

- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output.
- For example, take $c = a + b$. The key question is: if we perturb a by a small amount ϵ, how much does the output c change? In this case, the output c is also perturbed by 1ϵ, so the gradient (partial derivative) is 1. We put this gradient on the edge.
- We can handle $c = a \cdot b$ in a similar way.
- Intuitively, the gradient is a measure of local sensivity: how much input perturbations get amplified when they go through the various functions.



Basic building blocks

$$\begin{tikzcd} & \boxed{+} & \\ a \arrow[ur, "1"] & & b \arrow[ul, "1"'] \end{tikzcd}$$

$$\begin{tikzcd} & \boxed{-} & \\ a \arrow[ur, "1"] & & b \arrow[ul, "-1"'] \end{tikzcd}$$

$$\begin{tikzcd} & \boxed{\cdot} & \\ a \arrow[ur, "b"] & & b \arrow[ul, "a"'] \end{tikzcd}$$

$$\begin{tikzcd} \boxed{(\cdot)^2} \\ a \arrow[u, "2a"] \end{tikzcd}$$

$$\begin{tikzcd} & \boxed{\text{max}} & \\ a \arrow[ur, "\mathbf{1}[a > b]"] & & b \arrow[ul, "\mathbf{1}[a < b]"'] \end{tikzcd}$$

$$\begin{tikzcd} \boxed{\sigma} \\ a \arrow[u, "\sigma(a)(1 - \sigma(a))"] \end{tikzcd}$$

- Here are some more examples of simple functions and their gradients. Let's walk through them together.
- These should be familiar from basic calculus. All we've done is present them in a visually more intuitive way.
- For the max function, changing a only impacts the max iff $a > b$; and analogously for b.
- For the logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$, a bit of algebraic elbow grease produces the gradient. You can check that the gradient is zero when $|a| \to \infty$.
- It turns out that these simple functions are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter.

Function composition

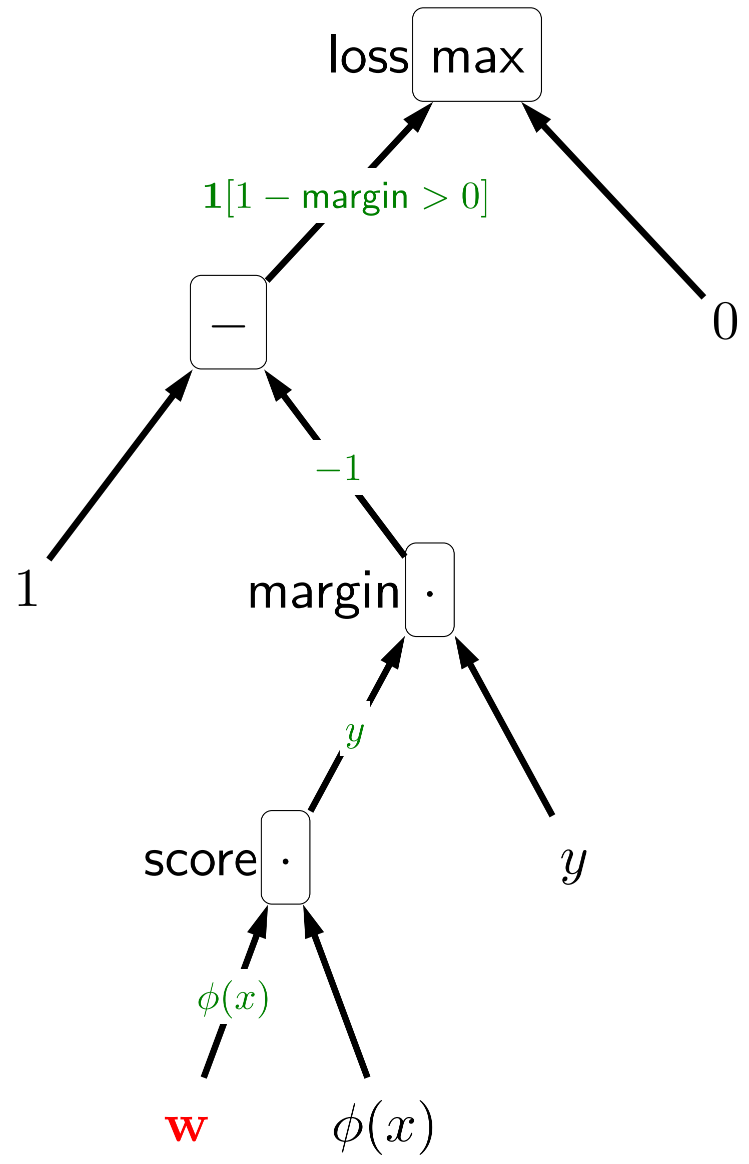
$$\begin{tikzcd}
c \; \boxed{(\cdot)^2} \\
b \; \boxed{(\cdot)^2} \arrow[u, "\frac{\partial c}{\partial b} = 2b"] \\
a \arrow[u, "\frac{\partial b}{\partial a} = 2a"]
\end{tikzcd}$$

Chain rule:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a} = (2b)(2a) = (2a^2)(2a) = 4a^3$$

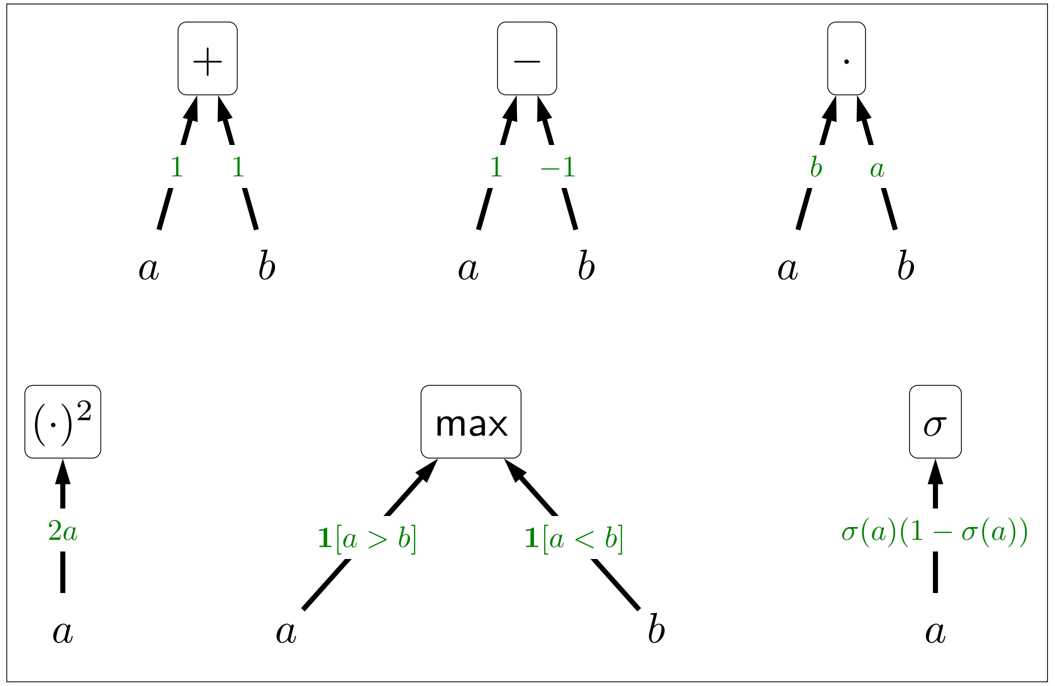
- Given these building blocks, we can now put them together to create more complex functions.
- Consider applying some function (e.g., squared) to a to get b, and then applying some other function (e.g., squared) to get c.
- What is the gradient of c with respect to a?
- We know from our building blocks the gradients on the edges.
- The final answer is given by the **chain rule** from calculus: just multiply the two gradients together.
- You can verify that this yields the correct answer $(2b)(2a) = 4a^3$.
- This visual intuition will help us better understand more complex functions.

Linear classification with hinge loss



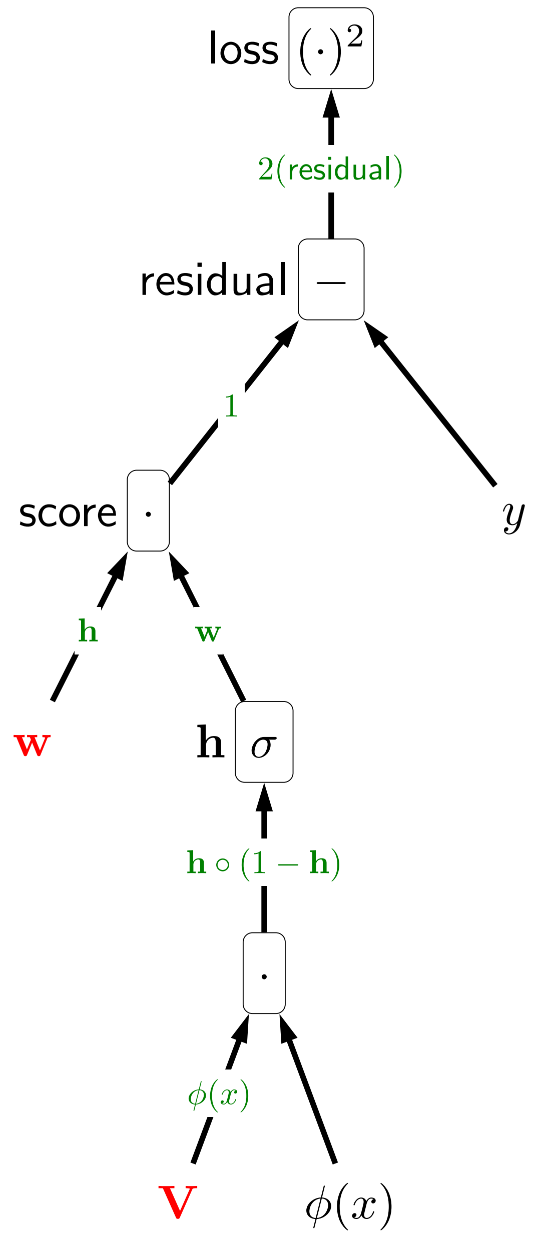
$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x) y, 0\}$$

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = -\mathbf{1}[\text{margin} < 1] \phi(x) y$$



- Now let's turn to our first real-world example: the hinge loss for linear classification. We already computed the gradient before, but let's do it using computation graphs.
- We can construct the computation graph for this expression, proceeding bottom up. At the leaves are the inputs and the constants. Each internal node is labeled with the operation (e.g., $\cdot$) and is labeled with a variable naming that subexpression (e.g., margin).
- In red, we have highlighted the weights $\mathbf{w}$ with respect to which we want to take the gradient. The central question is how small perturbations in $\mathbf{w}$ affect a change in the output (loss).
- We can examine each edge from the path from $\mathbf{w}$ to loss, and compute the gradient using our handy reference of building blocks.
- The actual gradient is the product of the edge-wise gradients from $\mathbf{w}$ to the loss output.

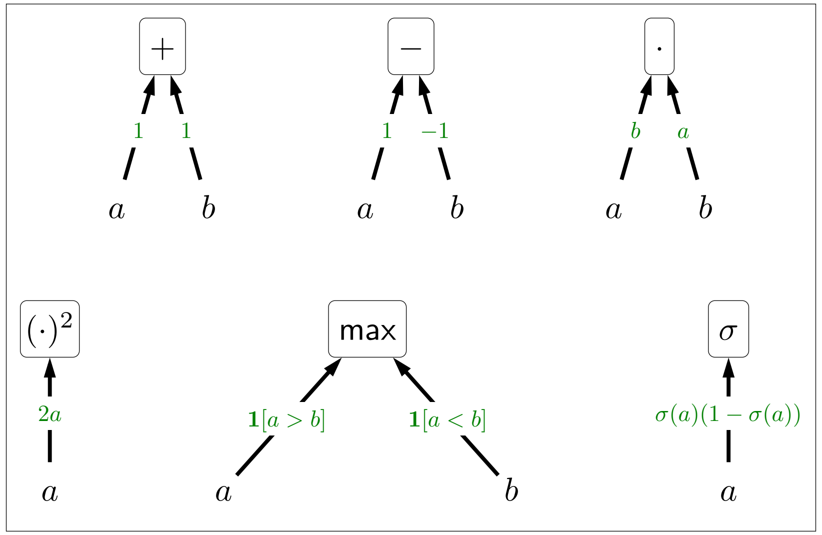
Two-layer neural networks



$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)) - y)^2$$

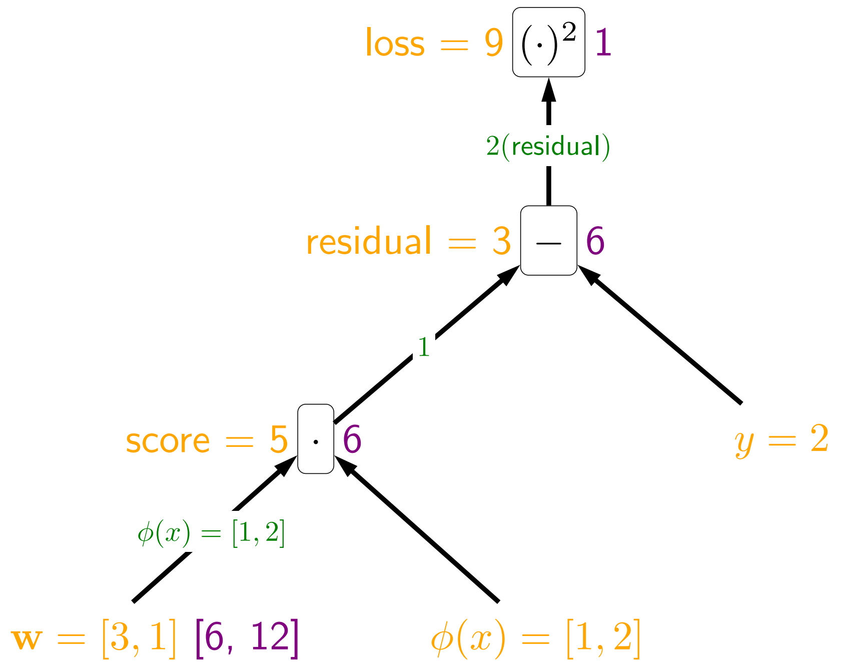
$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

$$\nabla_{\mathbf{V}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{w} \circ \mathbf{h} \circ (1-\mathbf{h})\phi(x)^\top$$



- We now finally turn to neural networks, but the idea is essentially the same.
- Specifically, consider a two-layer neural network driving the squared loss.
- Let us build the computation graph bottom up.
- Now we need to take the gradient with respect to $\mathbf{w}$ and $\mathbf{V}$. Again, these are just the product of the gradients on the paths from $\mathbf{w}$ or $\mathbf{V}$ to the loss node at the root.
- Note that the two gradients have in common the the first two terms. Common paths result in common subexpressions for the gradient.
- There are some technicalities when dealing with vectors worth mentioning: First, the $\circ$ in $\mathbf{h} \circ (1 - \mathbf{h})$ is elementwise multiplication (not the dot product), since the non-linearity σ is applied elementwise. Second, there is a transpose for the gradient expression with respect to $\mathbf{V}$ and not $\mathbf{w}$ because we are taking $\mathbf{V}\phi(x)$, while taking $\mathbf{w} \cdot \mathbf{h} = \mathbf{w}^\top \mathbf{h}$.
- This computation graph also highlights the modularity of hypothesis class and loss function. You can pick any hypothesis class (linear predictors or neural networks) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

Backpropagation



$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$

backpropagation

$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$

Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

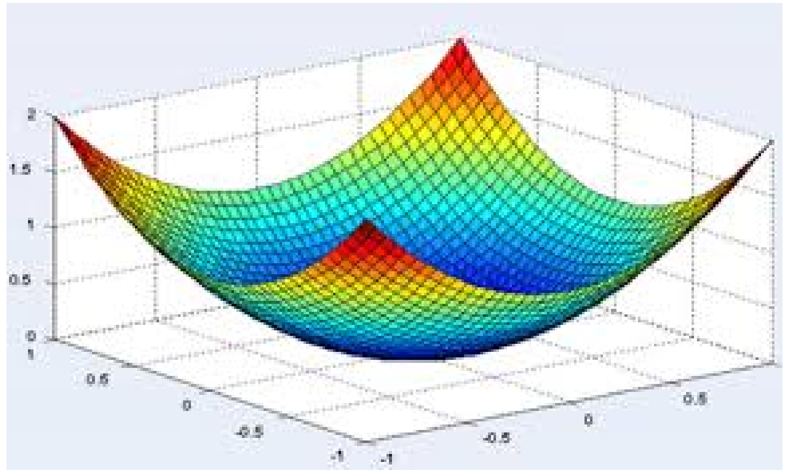
Backward pass: compute each g_i (from root to leaves)

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding.
- Now let us introduce the **backpropagation** algorithm, a general procedure for computing gradients given only the specification of the function.
- Let us go back to the simplest example: linear regression with the squared loss.
- All the quantities that we've been computing have been so far symbolic, but the actual algorithm works on real numbers and vectors. So let's use concrete values to illustrate the backpropagation algorithm.
- The backpropagation algorithm has two phases: forward and backward. In the forward phase, we compute a **forward value** f_i for each node, coresponding to the evaluation of that subexpression. Let's work through the example.
- In the backward phase, we compute a **backward value** g_i for each node. This value is the gradient of the loss with respect to that node, which is also the product of all the gradients on the edges from the node to the root. To compute this backward value, we simply take the parent's backward value and multiply by the gradient on the edge to the parent. Let's work through the example.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but have the same dimensionality.

A note on optimization

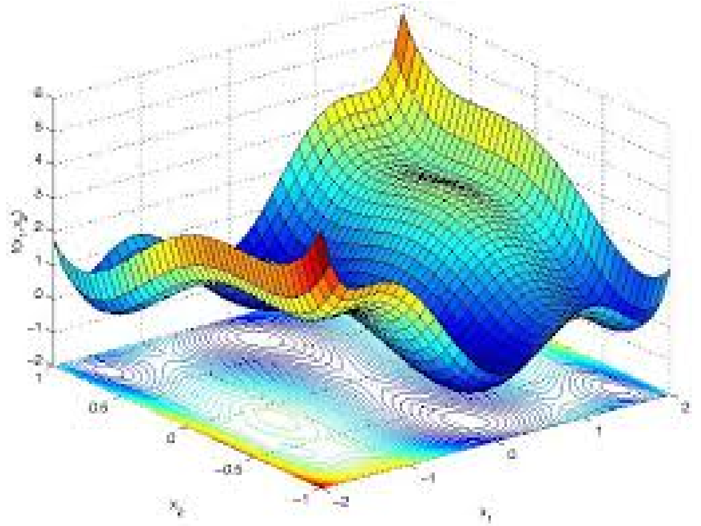
$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

Neural networks

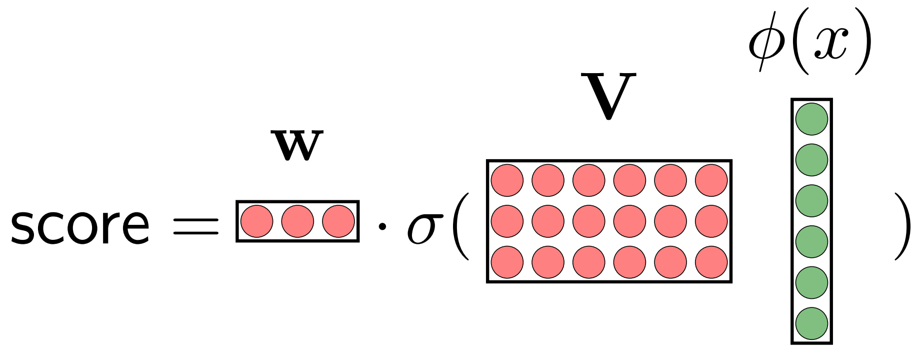


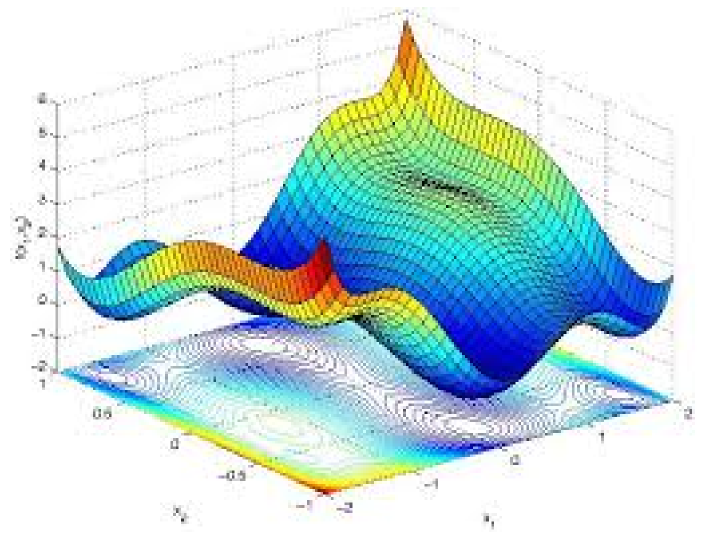
(non-convex)

Optimization of neural networks is in principle hard

- So now we can apply the backpropagation algorithm and compute gradients, stick them into stochastic gradient descent, and get some answer out.
- One question which we haven't addressed is whether stochastic gradient descent will work in the sense of actually finding the weights that minimize the training loss?
- For linear predictors (using the squared loss or hinge loss), $\text{TrainLoss}(\mathbf{w})$ is a convex function, which means that SGD (with an appropriately step size) is theoretically guaranteed to converge to the global optimum.
- However, for neural networks, $\text{TrainLoss}(\mathbf{V}, \mathbf{w})$ is typically non-convex which means that there are multiple local optima, and SGD is not guaranteed to converge to the global optimum. There are many settings that SGD fails both theoretically and empirically, but in practice, SGD on neural networks can work much better than theory would predict, provided certain precautions are taken. The gap between theory and practice is not well understood and an active area of research.

How to train neural networks



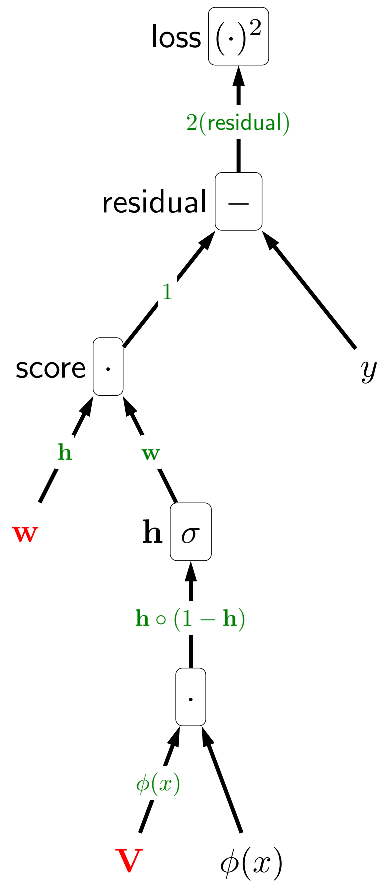


- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don't let gradients vanish or explode!

- Training a neural network is very much like driving stick. In practice, there are some "tricks" that are needed to make things work properly. Just to name a few to give you a sense of the considerations:
- Initialization (where you start the weights) matters for non-convex optimization. Unlike for linear models, you can't start at zero or else all the subproblems will be the same (all rows of $\mathbf{V}$ will be the same). Instead, you want to initialize with a small amount of random noise.
- It is common to use overparameterized neural networks, ones with more hidden units (k) than is needed, because then there are more "chances" that some of them will pick out on the right signal, and it is okay if some of the hidden units become "dead".
- There are small but important extensions of stochastic gradient descent that allow the step size to be tuned per weight.
- Perhaps one high-level piece of advice is that when training a neural network, it is important to monitor the gradients. If they vanish (get too small), then training won't make progress. If they explode (get too big), then training will be unstable.

Summary



- Computation graphs: visualize and understand gradients
- Backpropagation: general-purpose algorithm for computing gradients

- The most important concept in this module is the idea of a **computation graph**, allows us to represent arbitrary mathematical expressions, which can just be built out of simple building blocks. They hopefully have given you a more visual and better understanding of what gradients are about.
- The **backpropagation** algorithm allows us to simply write down an expression, and never have to take a gradient manually again. However, it is still important to understand how the gradient arises, so that when you try to train a deep neural network and your gradients vanish, you know how to think about debugging your network.
- The generality of computation graphs and backpropagation makes it possible to iterate very quickly on new types of models and loss functions and opens up a new paradigm for model development: **differential programming**.