Lecture 2: Backpropagation

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

Today's lecture will be entirely devoted to the backpropagation algorithm. The heart of all deep learning.

THE PLAN

part 1: review

part 2: scalar backpropagation

part 3: tensor backpropagation

part 4: automatic differentiation

VU

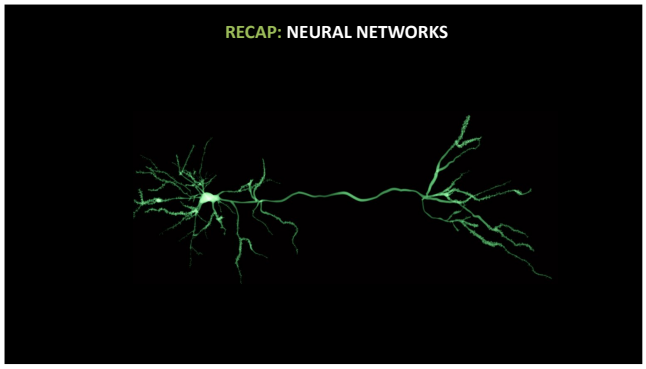
In **the first part**, we will review the basics of neural networks, and describe backpropagation in a scalar setting. That is, we will treat each individual element as a single number, and simply loop over all these numbers to do backpropagation over the whole network. This simplifies the derivation, but it is ultimately a slow algorithm with a complicated notation.

In **the second part**, we translate neural networks to operations on vectors, matrices and tensors (the higher-dimensional analogue of a matrix). This allows us to simplify our notation, and more importantly, massively speed up the computation of neural networks. Backpropagation on tensors is a little more difficult to do than backpropagation on scalars.

In **the third part**, we will make the final leap from manually worked out and implemented backpropagation system to full-fledged automatic differentiation: we will show you how to build a system that take care of the gradient computation entirely by itself. This is the technology behind software like pytorch and tensorflow.

PART ONE: REVIEW

VU

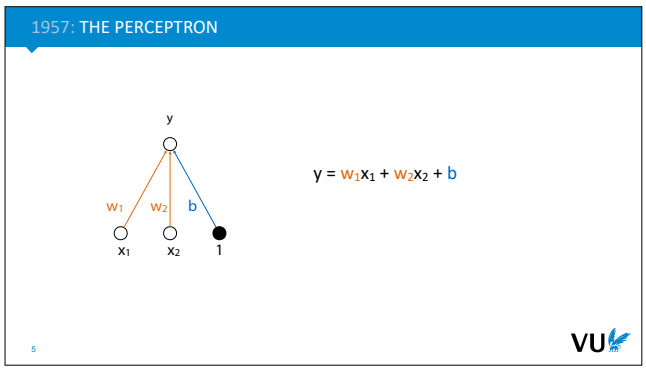


We'll start with a quick recap of the basic principles behind neural networks. The name neural network is a bit of a historical artifact.

In the very early days of AI (the late 1950s), researchers decided to take a simple approach to AI. They started with a single brain cell: a neuron. A neuron receives multiple different signals from other cells through connections called **dendrites**. It processes these in a relatively simple way, deriving a single new signal, which it sends out through its single **axon**. The axon branches out so that the single signal can reach other cells.

image source: http://www.sciencealert.com/scientists-build-an-artificial-neuron-that-fully-mimics-a-human-brain-cell

---



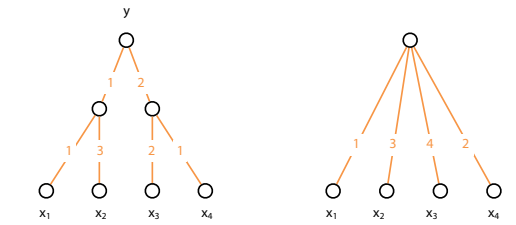
This principles needed to be radically simplified to work with computers of that age, but doing so yielded one of the first successful machine learning systems: the perceptron (also seen in the **video** in the first lecture).

The perceptron had a number of inputs (the features in modern parlance), each of which was multiplied by a weight. These result was summed, together with a bias parameter, and the sign of this result was used for sclassification or regression.

Note that the intercept can be represented as just another input that we just fix to always be 1. This is called a **bias node**.

Of course, there is nothing new here: this is just a basic linear regression or linear classification model. The real power of the brain is in chaining multiple neurons together in a network.

PROBLEM: COMPOSING NEURONS



$y = 1(1x_1 + 3x_2) + 2(2x_3 + 1x_4)$

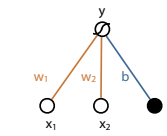
$y = 1x_1 + 3x_2 + 4x_3 + 2x_4$

6

This is where the perceptron turns out to be too simple an abstraction. Because composing perceptrons (making the output of one perceptron the input of another) doesn't make it more powerful. All you end out with is something that is equivalent to another linear model. We're not creating models that can learning non-linear functions.

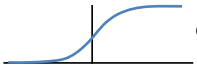
We've removed the bias node here for clarity, but that doesn't affect our conclusions: any composition of affine functions is itself an affine function.

NONLINEARITY



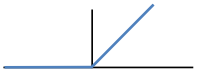
$$y = \sigma(w_1 x_1 + w_2 x_2 + b)$$

sigmoid



$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

ReLU



$$r(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

7

The simplest solution is to apply a nonlinear function to each neuron, called the **activation function.** This is a scalar function we apply to the output of a perceptron after all the weighted inputs have been combined.

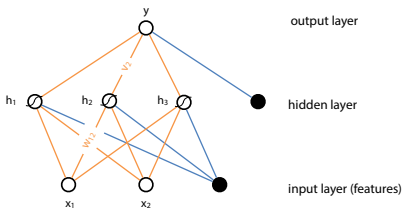
One popular option (especially in the early days) is the **logistic sigmoid**, which we've seen already. Applying a sigmoid means that the sum of the inputs can range from negative infinity to positive infinity, but the output is always in the interval [0, 1].

Another, more recent nonlinearity is the linear rectifier, or ReLU nonlinearity. This function just sets every negative input to zero, and keeps everything else the same.

Not using an activation function is also called using a linear activation.

We've seen one interpretation of the sigmoid function already: it turns every perceptron into a logistic classification unit. For now, we won't look much more into the meaning of the nonlinearities: we know we need them to make a stack of neuron more expressive than a single neuron, and by trial and error, we've found a few that work well for us.

FEEDFORWARD NETWORK



aka Multilayer Perceptron (MLP)

8

Using these nonlinearities, we can arrange single neutrons into **neural networks**. Any arrangement makes a neural network, but for ease of training, this arrangement was the most popular for a long time. It's called the **feedforward network** or **multilayer perceptron**. We arrange a layer of hidden units in the middle, each of which acts as a perceptron with a nonlinearity, connecting to all input nodes. Then we have one or more output nodes, connecting to all hidden layers. Crucially:

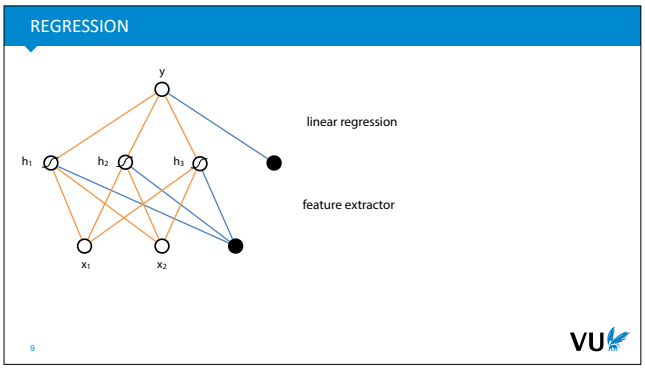
- There are no cycles, the network feeds forward from input to output.
- Nodes in the same layer are not connected to each other, or to any other layer that the

previous one.

- Each layer is fully connected to the previous layer, every node in one layer connects to every node in the layer before it.

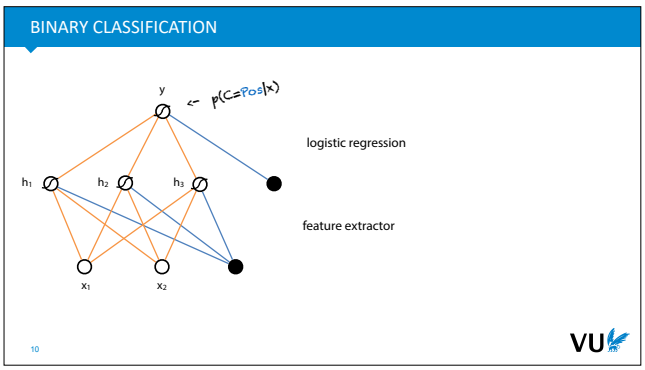
In the 80s and 90s they usually had just one hidden layer, because we hadn't figured out how to train deeper networks.

Every orange and blue line in this picture represents one parameter of the model.

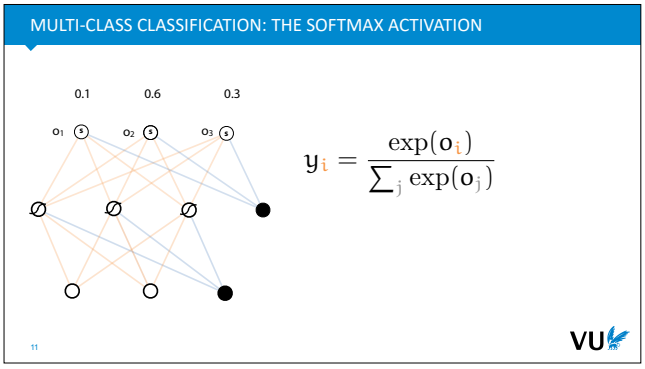


If we want to train a regression model, we put non-linearities on the hidden nodes, and no activation on the output node. That way, the output can range from negative to positive infinity.

We can think of the first layer as learning some nonlinear transformation of the features, and the second layer as performing linear regression on the features.



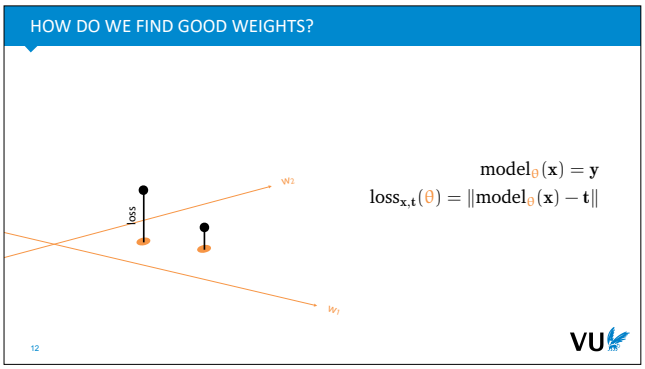
If we have a classification problem with two classes (a binary classification problem), we can place a sigmoid activation on the output layer, so that the output is between 0 and 1. We can then interpret this as the probability that the input has the **positive** class.



For multi-class classification, we can use the **softmax activation**. We create a single output node per class, and ensure that they sum to one. We can then interpret the output of the network as **class probabilities**.

This activation is a little unusual in that it's not strictly element-wise: to compute the value of one output node, it looks at the inputs of all the other outputs nodes. To compute it we simply take the exponent of each output node (to ensure that they are all positive) and then divides each by the total (to ensure that they sum to one).

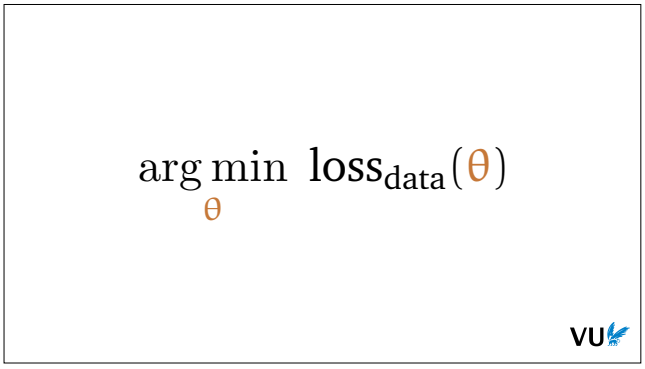
After the softmax we can interpret the output of node y_3 as the probability that **x** has class 3, and train with cross entropy loss.



To find good weights we first define a loss function. This is a function of a particular model (represented by its the weights) to a scalar value. **The better our model, the lower the loss.** If we imagine a model with just two weights, the **model space** forms a plane. For every point in this plane, our loss function defines a scalar loss, which we can draw above the plane as a surface: the **loss surface** (sometimes also called, more poetically, the loss landscape).

Make sure you understand the difference between the model, a function from the inputs x to the outputs y in which the parameters theta act as constants, and the loss, a function from the parameters to a loss value, in which the data acts as constants.

The symbol θ is a common notation referring to the set of all weights (sometimes combined into a vector, sometimes just a set).



SOME COMMON LOSS FUNCTIONS

regression	squared errors	$\|\mathbf{y} - \mathbf{t}\|$ with $\mathbf{y} = \text{model}_\theta(\mathbf{x})$
	absolute errors	$\|\mathbf{y} - \mathbf{t}\|_1 = \sum_i \text{abs}(y_i - t_i)$
classification	binary cross-entropy	$-\log p_\theta(t)$ with $t \in \{0, 1\}$
	cross-entropy	$-\log p_\theta(t)$ with $t \in \{0, \ldots, K\}$
	hinge loss	$\max(0, 1 - ty)$ with $t \in \{-1, 1\}$

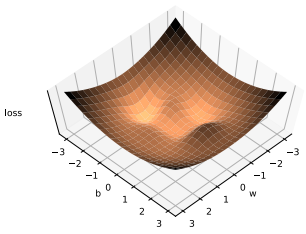
VU

Here are some common loss functions for situations where we have examples (t) of what the model output (y) should be for a given input (x).

The squared error losses are derived from basic regression. The (binary) cross entropy comes from logistic regression (as shown last lecture) and the hinge loss comes from support vector machine classification. You can find their derivations in most machine learning books/courses. We won't elaborate on them here, except to note that in all cases the loss is lower if the model output (y) is closer to the example output (t).

The loss can be computed for a single example or for multiple examples. **In almost all cases, the loss for multiple examples is just the sum over all their individual losses.**

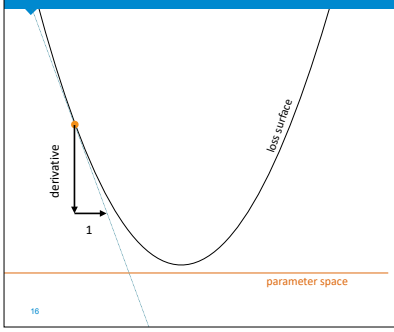
LOSS SURFACE



VU

We want to follow the loss surface down to the lowest point.

DERIVATIVE

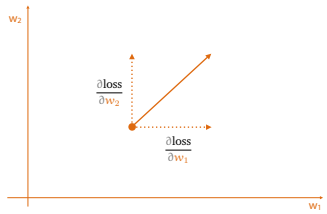


VU

In one dimension, we know that the derivative of a function (like the loss) tells us how much a function increases or decreases in we take a step of size 1 to the right.

To be more precise, it tells us how much the best linear approximation (the tangent line) increases or decreases.

GRADIENT



$$\nabla_\theta \text{loss}_{\mathbf{x},\mathbf{t}}(\theta)$$

If our input space has multiple dimensions, like our loss surface, we can simply take the derivative with respect to each input, seprately, treating the others as constants. This is called a **partial derivative**. The collection of all possible partial derivatives is called **the gradient**.

If we interpret the gradient as a vector, it points in the direction in which the function grows the fastest. Taking a step in the opposite direction means we are walking down the loss surface.

The symbol for the gradient is a downward pointing triangle called a nabla. The subscript indicates the variable over which we are taking the derivatives. Note that in this case we are treating

STOCHASTIC GRADIENT DESCENT

pick some initial weights θ (for the whole model)

loop: ← epochs

for **x, t** in **Data**:

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta \text{loss}_{\mathbf{x},\mathbf{t}}(\theta)$$

learning rate

stochastic gradient descent: loss over one example per step (loop over data)

minibatch gradient descent: loss over a few examples per step (loop over data)

full-batch gradient descent: loss over the whole data

This is the idea behind the gradient descent algorithm. We compute the gradient, take a small step in the opposite direction and repeat. The reason we take small steps is that the gradient is only the direction of steepest ascent locally, the further we move from our current position the worse an approximation the tangent hyperplane will be for the function that we are actually trying to follow.

In deep learning, we almost always use **minibatch gradient descent,** but there are some examples.

RECAP

perceptron: linear combination of inputs

neural network: network of perceptrons, with scalar nonlinearities

training: minibatch gradient descent

But, how do we compute the gradient of a complex neural network?
Next video: **backpropagation**.

Lecture 2: Backpropagation

Peter Bloem
Deep Learning 2020

dlvu.github.io

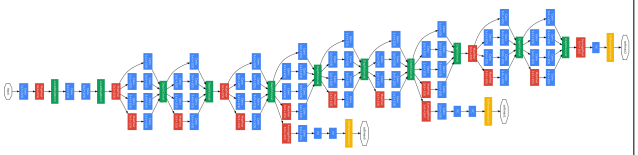
VRIJE UNIVERSITEIT AMSTERDAM

PART TWO: **SCALAR BACKPROPAGATION**

How do we work out the gradient for a neural network?

VU

Working out a gradient is usually done by hand, with pen and paper. This function is then transferred to code, and used in a gradient descent loop. But the more complicated out model becomes, the more complex it becomes to work out a complete formulation of the gradient.

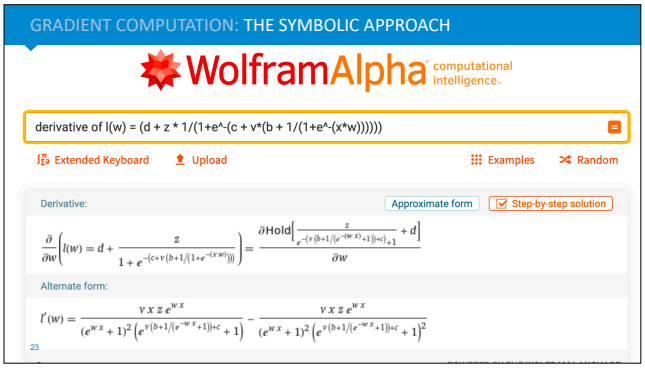


VU

For very simple models like linear regression or logistic regression, we can just work out the gradient by hand, with pen and paper.

Hewever, here is a diagram of the sort of network we'll be encountering (the GoogLeNet). We can't work out a complete gradient for this kind of architecture by hand. We need help.

GRADIENT COMPUTATION: THE SYMBOLIC APPROACH



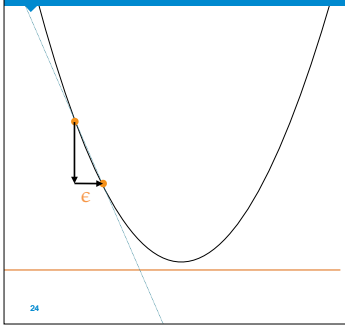
23

Of course, working out derivatives is a pretty mechanical process. We could easily take all the rules we know, and put them into some algorithm. This is called symbolic differentiation, and it's what systems like Wolfram Alpha do for us.

Unfortunately, as you can see here, the gradients it returns get pretty horrendous the deeper the neural network gets. This approach becomes impractical very quickly.

Note that in symbolic differentiation we get an answer that **is independent of the input**. We get a function that we can then feed and input to.

GRADIENT COMPUTATION: THE NUMERIC APPROACH



$$\text{deriv} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

the "method of finite differences"

24

VU

Another approach is to compute the gradient numerically. For instance by the method of finite differences: we take a small step ε and, see how much the function changes. The amount of change divided by the step size is a good estimate for the gradient if ε is small enough.

Numeric approaches are sometimes used, but it's very expensive to make them accurate if you have a large number of parameters.

Note that in the numeric approach, you only get an answer **for a particular input**. If you want to compute the gradient at some other point in space, you have to compute again.

BACKPROPAGATION: THE MIDDLE GROUND

Work out parts of the derivative **symbolically**
chain these together in a **numeric computation**.

secret ingredient: **the chain rule**.

Backpropagation is a kind of middle ground between symbolic and numeric gradient descent.

THE CHAIN RULE

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

shorthand:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

The single thing that makes backpropagation possible is *the chain rule of differentiation*. If we want the derivative of a function which is the composition of two other function, in this case f and g, we can take the derivative of f with respect to the output of g and multiply it by the derivative of g with respect to the input x.

Since we'll be using the chain rule *a lot*, we'll introduce a simple shorthand to make it a little easier to parse. We draw a little diagram of which function feeds into which. This means we know what the argument of each function is, so we can remove the arguments from our notation.

INTUITION FOR THE CHAIN RULE

slope of f over g

$$f(g) = s_f g + b_f$$
$$g(x) = s_g x + b_g$$

$$\text{with } s_f = \frac{\partial f}{\partial g}$$
$$s_g = \frac{\partial g}{\partial x}$$

$$f(g(x)) = s_f(s_g x + b_g) + b_f$$
$$= s_f s_g x + s_f b_g + s_g$$

slope of f over x — constant

$$\frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

Since the chain rule is the heart of backpropagation, and backpropagation is the heart of deep learning, we should probably take some time to see why the chain rule is true at all.

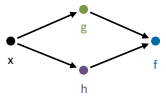
If we imagine that f and g are linear functions, it's pretty straightforward to show that this is true. They may not be, of course, but the nice thing about calculus is that locally, we can treat them as linear functions (if they are differentiable). In an infinitessimally small neighbourhood f and g are exactly linear.

In that case f and g both have a *slope,* s_f and s_g, which is simply the derivative at x. They also have additive constants b_f and b_g, which we're not interested in. For these linear approximations, we can easily work out what what the function f(g(x)) looks like and what its slope is. It's the slope of f times the slope of g (plus some constant value that doesn;t depend on x). This is exactly what the chain rule tells us.

If you want an example: imagine you have one investment scheme that doubles your money, and one that triples it: then, if you take the outcome of the first and put it into the second, you are multiplying your money by 6.

Note that this doesn't quite count as a proof, but it's hopefully enough to give you some intuition for why the chain rule holds.

MULTIVARIATE CHAIN RULE



$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

VU

Since we'll be looking at some pretty elaborate compution graphs, we'll need to be able to deal with this situation as well. We have a computation graph, as before, but f depends on x through two different operations. How do we take the derivative of f over x?

The multivariate chain rule tells us that we can simply apply the chain rule along g, taking h as a constant, and sum it with the chain rule along h taking g as a constant.

INTUITION FOR THE MULTIVARIATE CHAIN RULE

$$f(g,h) = s_1 g + s_2 h + b_f$$
$$g(x) = s_g x + b_g$$
$$h(x) = s_h x + b_h$$

with $s_1 = \frac{\partial f}{\partial g}$ $s_2 = \frac{\partial f}{\partial h}$ $s_g = \frac{\partial g}{\partial x}$ $s_h = \frac{\partial h}{\partial x}$

$$\begin{aligned} f(g(x),h(x)) &= s_1(s_g x + b_g) + s_2(s_f x + b_f) + b_f \\ &= s_1 s_g x + s_1 b_g + s_2 s_f x + s_2 b_f + b_f \\ &= (s_1 s_g + s_1 s_h)x + s_2 b_g + s_2 b_f + b_f \end{aligned}$$

slope of f over x — constant

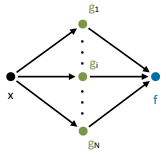
$$\frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}$$

VU

We can see why this holds in the same way as before. The short story, since all functions can be taken to be linear, their slopes distribute out into a sum

---

MULTIVARIATE CHAIN RULE



$$\frac{\partial f}{\partial x} = \sum_i \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$

VU

If we have more than two paths from the input to the output, we simply sum over all of them.

EXAMPLE

$$f(x) = \frac{2}{\sin(e^{-x})}$$

VU

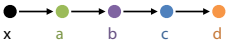
ITERATING THE CHAIN RULE

$$f(x) = \frac{2}{\sin(e^{-x})} \qquad f(x) = d(c(b(a(x))))$$

operations:

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^{a}$$
$$a(x) = -x$$

computation graph:



VU

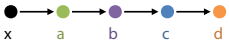
Here's an example of how the chain rule can help us to automate complicated derivatives. We start with the function on the top left. We break up its form into a series of smaller operations. The entire function f is then just a chain of these small operations chained together. We can draw this in a diagram as we did before.

We will call this kind of diagram a **computation graph**.

Normally, we wouldn't break a function up in such small operations. This is just a simple example to illustrate the principle.

ITERATING THE CHAIN RULE

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial d}{\partial x} \\ &= \frac{\partial d}{\partial c}\frac{\partial c}{\partial x} \\ &= \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial x} \\ &= \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\end{aligned}$$



VU

Now, to work out the derivative of f, we can *iterate the chain rule*. We apply it again and again, until the derivative of f over x is expressed as a long product of derivatives of operation outputs over their inputs.

GLOBAL AND LOCAL DERIVATIVES

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

global derivative — local derivatives

VU

We call the larger derivative of f over x the **global derivative**. And we call the individual factors, the derivatives of the operation output wrt to their inputs, the local derivatives.

BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them

VU

WORK OUT THE LOCAL DERIVATIVES SYMBOLICALLY

$$f(x) = \frac{2}{\sin(e^{-x})}$$

operations:

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^{a}$$
$$a(x) = -x$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$

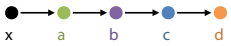
$$= -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

VU

Fir each local derivative, we work out, on pen an paper the symbolic derivative. Note that we could fill in the a, b and c in the result, but we don't. We simply leave them as is, and continue with the numerical part of the algorithm.

COMPUTE A FORWARD PASS (X = -4.499)

$$f(-4.499) = 2$$



$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^{a} = 90$$
$$a = -x = 4.499$$

VU

We start by computing what is known as a forward pass. We pick some input, in this case x = -4.499, and we feed it through the computation graph. Crucially, we save all the intermediate values, as well as the output.

Note that at this point, we are no longer computing solutions in general. We are computing our function for a specific input. We'll be computing the gradient for this specific input as well.

COMPUTE A FORWARD PASS (X = -4.499)

$$f(-4.499) = 2$$
$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^{a} = 90$$
$$a = -x = 4.499$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^{a} \cdot -1$$

$$= -\frac{2}{1^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$

$$= -2 \cdot 0 \cdot 90 \cdot -1 = 0$$

VU

Keeping all intermediate values from the forward pass in memory, we go back to our symbolic expression of the derivative. Here, we fill in the intermediate values a b and c. After we do this, we can finish the multiplication numerically, giving us a numeric value of the gradient of f at x = -4.499.

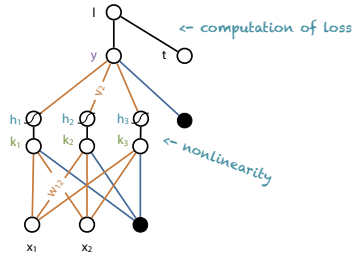
BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them
- Much more accurate than finite differences
  only source of inaccuracy is the numeric computation of the operations.
- Much faster than symbolic differentiation
  The backward pass has (broadly) the sample complexity as the forward.

VU

BRACKPROPAGATION IN A FEEDFORWARD NETWORK



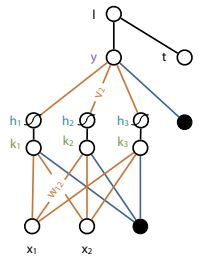
40

To explain how backpropagation works in a neural network, we extend our diagram a little bit, to make it closer to the actual computation graph we'll be using.

First, we separate the hidden node into the result of the linear operation k_i and the application of the nonlinearity h_i. Second, since we're interested in the derivative of the loss rather than the output of the network, we extend the network with one more step: the computation of the loss (over one example to keep things simple). In this final step, the output y of the network is compared to the target value t from the data, producing a loss value.

You can thin of t as another input node, like x_1 and x_2, but one to which the model doesn't have access.

BRACKPROPAGATION IN A FEEDFORWARD NETWORK



$$\frac{\partial l}{\partial v_2}, \frac{\partial l}{\partial w_{12}}$$

$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$h_2 = \frac{1}{1 + \exp(-k_2)}$$

$$k_2 = w_{12} x_1 + w_2 x_2 + b_x$$

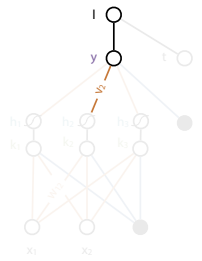
41

We want to work out the gradient of the loss over the parameters. We'll isolate two parameters, v_2 in the second layer, and w_{12} in the first, and see how backpropagation operates.

First, we have to break the computation of the loss into operations. If we take the graph on the left to be our computation graph, then we end up with the operations of the right.

To simplify things, we'll compute the loss over only one example. We'll also use a simple squared error loss.

BACKPROPAGATION IN A FEEDFORWARD NETWORK



$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial v_2}$$

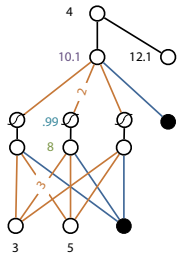
$$= 2(y - t) \cdot h_2$$

42

For the derivative with respect to v_2, we'll only need these two operations. Anything below doesn't affect the result.

To work out the derivative we apply the chain rule, and work out the local derivatives symbolically.

BACKPROPAGATION IN A FEEDFORWARD NETWORK



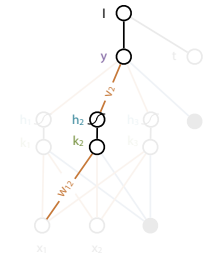
$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial v_2}$$
$$= 2(y - t) \cdot h_2$$
$$= 2(10.1 - 12.1) \cdot .99 = -3.96$$
$$v_2 \leftarrow v_2 - \alpha \cdot -3.96$$

We then do a forward pass with some values. We get an output of 10.1, which should have been 12.1, so our loss is 4. We keep all intermediate values in memory.

We then take our product of local derivatives, fill in the numeric values from the forward pass, and compute the derivative over v_2.

When we apply this derivative in a gradient descent update, v_2 changes as shown below.

BACKPROPAGATION IN A FEEDFORWARD NETWORK



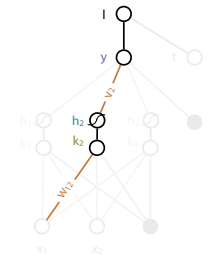
$$l = (y - t)^2$$
$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$
$$h_2 = \frac{1}{1 + \exp(-k_2)}$$
$$k_2 = w_{12} x_1 + w_2 x_2 + b_x$$

$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2}\frac{\partial k_2}{\partial w_{12}}$$
$$= 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

derivative of sigmoid: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Let's try something a bit earlier in the network: the weight w_{12}. We add two operations, apply the chain rule and work out the local derivatives.

BACKPROPAGATION



$$\frac{\partial l}{\partial y}$$
$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2} = \frac{\partial l}{\partial h_2}$$
$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2} = \frac{\partial l}{\partial k_2}$$
$$\frac{\partial l}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial k_2}\frac{\partial k_2}{\partial w_{12}} = \frac{\partial l}{\partial w_{12}}$$

Note that when we're computing the derivative for w_12, we are also, along the way computing the derivatives for y, h2 and k2.

This useful when it comes to implementing backpropagation. We can walk backward from the computation graph and compute the derivative of the loss for every node. For the nodes below, we just multiply the local gradient. This means we can very efficiently compute any derivatives we need.

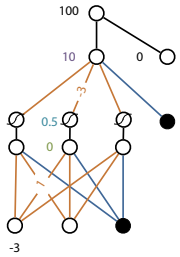
We will show this more precisely in the last part of this lecture.

BACKPROPAGATION

The **BACKPROPAGATION** algorithm:

- break your computation up into a sequence of operations
  what counts as an operation is up to you
- work out the **local derivatives** symbolically.
- compute the **global derivative** numerically
  by computing the local derivatives and multiplying them
- Walk backward from the loss, *accumulating* the derivatives.

BUILDING SOME INTUITION

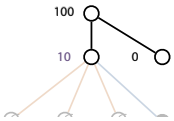


47

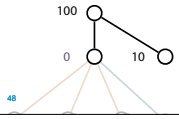
To finish up, let's see if we can build a little intuition for what all these accumulated derivatives mean.

Here is a forward pass for some weights and some inputs. Backpropagation starts with the loss, and walks down the network, figuring out at each step how every value contributed to the result of the forward pass. Every value that contributed positively to a positive loss should be lowered, every value that contributed positively to a negative loss should be increased, and so on.

IMAGINE THAT y IS A PARAMETER



$$y \leftarrow y - \alpha \cdot 2(y - t)$$
$$= y - \alpha \cdot 20$$



$$y \leftarrow y - \alpha \cdot 2(y - t)$$
$$= y + \alpha \cdot 20$$

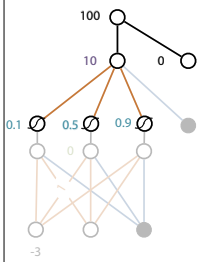
48

We'll start with the first value below the loss: y. Of course, this a isn't *parameter* of the network, but let's imagine for a moment that it is. What would the gradient descent update rule look like if we try to update y?

Even though we can't change y directly, this is the effect we want to achieve: we want to change the parameters we *can* change so that we achieve this change in y.

Note that the error (the derivative of the loss) takes **the sign** into account. If our model output is 0, and our target is 10, the loss is the same, but the error is *minus 20*, and the update rule for y

---



$$v_2 \leftarrow v_2 - \alpha \cdot 2(y - t)h_2$$

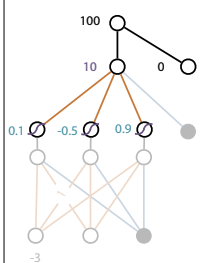
$$v_1 \leftarrow v_1 - \alpha \cdot 20 \cdot 0.1$$
$$v_2 \leftarrow v_2 - \alpha \cdot 20 \cdot 0.5$$
$$v_3 \leftarrow v_3 - \alpha \cdot 20 \cdot 0.9$$

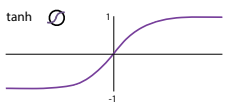
49

Instead of changing y, we have to change the values that influenced y. Here we see what that looks like for the weights of the second layer.

Note that the current value of the weight doesn't factor into the update. Only how much influence the weight had on the value of y in the forward pass. The higher the activation of the source node, the more the weight gets adjusted.

NEGATIVE ACTIVATION





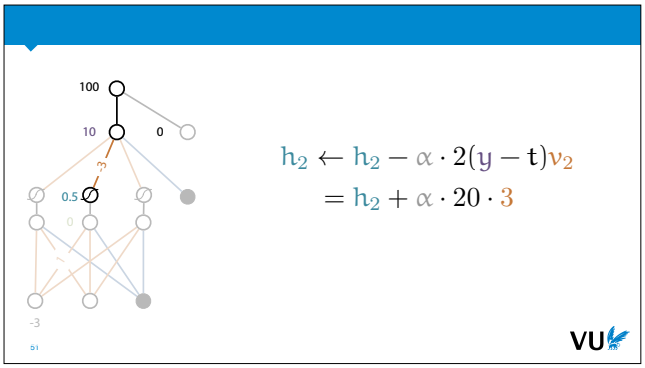
$$v_1 \leftarrow v_1 - \alpha \cdot 20 \cdot 0.1$$
$$v_2 \leftarrow v_2 + \alpha \cdot 20 \cdot 0.5$$
$$v_3 \leftarrow v_3 - \alpha \cdot 20 \cdot 0.9$$

50

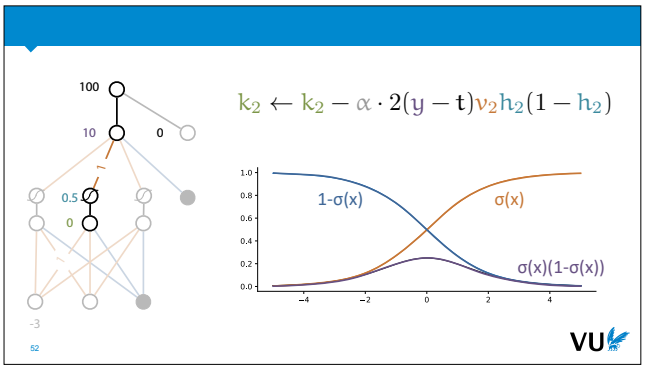
The sigmoid activation we've used so far allows only positive values to emerge from the hidden layer. If we switch to an activation that also allows negative activations (like a linear activation or a **tanh** activation), we see that backpropagation very naturally takes the sign into account.

In this case, we want to update in such a way that y decreases, but we note that the weight v_2 is multiplied by a *negative* value. This means that (for this instance) v_2 contributes negatively to the loss, and its value should be increased.

Note that the sign of v_2 itself doesn't matter. Whether it's positive or negative, its value should increase.



If we could change h_2 directly, this is how we'd do it. We take the value of v_2 to be a constant. We want to decrease the output of the network. Since v_2 makes a *negative* contribution to the loss, we can achieve this by *increasing* the activation of the source node of v_2.

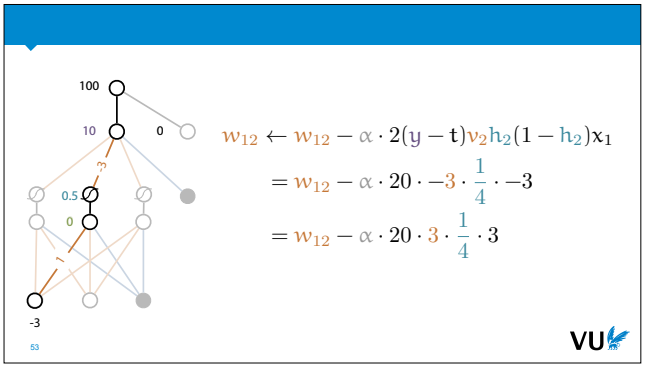


Moving down to k2, remember that the derivative of the sigmoid is the output of the sigmoid times 1 minus that output.

We see here, that in the extreme regimes, the sigmoid is *resistant to change*. The close to 1 or 0 we get the smaller the weight update becomes.

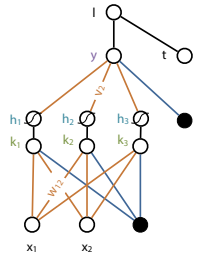
This is actually a great downside of the sigmoid activation, and one of the big reasons it was eventually replaced by the ReLU as the default choice for hidden units. We'll come back to this in later lectures.

Nevertheless, this update rule tells us what the change is to k_2 that we *want to achieve* by changing the gradients we can actually change (in this case the weights of layer 1).



Finally, we come to the weights of the first layer. As before, we want the output of the network to *de*crease. To achieve this, we want h_2 to *in*crease (because v_2 is negative). However, the input x_1 is negative, so we should decrease w_{12} to increase h_2. This is all beautifully captured by the chain rule: the two negatives of x_1 and v_2 cancel out and we get a positive value which we subtract from w_{12}.

FORWARD PASS IN PSEUDOCODE

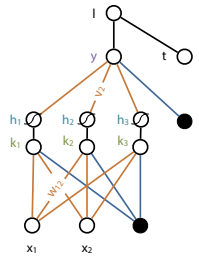



```
for j in [1 … 3]:
    for i in [1 … 2]:
        k[j] += w[i,j] * x[i]
    k[j] += b[j]

for i in [1 … 3]:
     h[i] = sigmoid(k[i]))

for i in [1 … 3]:
     y += h[i] * v[i]
y += c

loss = (y - t) ** 2
```


VU

Assume that k and y are initialized with 0s.

BACKWARD PASS IN PSEUDOCODE




```
y' = 2 * (y - t) # the error
for i in [1 … 3]:
     v'[i] = y' * h[i]
     h'[i] = y' * v[i]
c' = y'

for i in [1 … 3]:
    k'[i] = h'[i] * h[i] * (1-h[i])

for j in [1 … 3]:
    for i in [1 … 2]:
        w'[i, j] = k'[j] * x[i]
    b'[j] = k'[j]
```


VU

Note that we don't implement the derivations from slide 39 directly. Instead, we work backwards down the neural network: computing the derivative of each node as we go by taking the derivative of the los over the outputs and multiplying it by the local derivative.

RECAP

Backpropagation: method for computing derivatives.

Combined with gradient descent to train NNs.

Middle ground between symbolic and numeric computation.

- Break computation up into operations.
- Work out local derivatives symbolically.
- Work out global derivative numerically.

VU

Lecture 2: Backpropagation

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

PART THREE: **TENSOR BACKPROPAGATION**

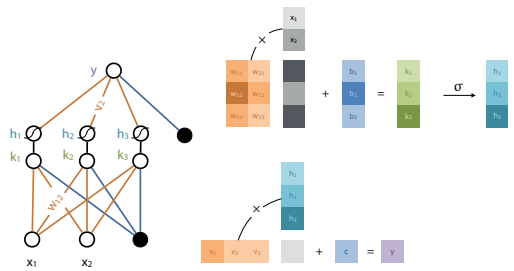
Expressing backpropagation in vector, matrix and tensor operations.

VU

We've seen what neural networks are, how to train them by gradient descent and how to compute that gradient by backpropagation.

In order to scale up to larger and more complex structures, we need two make our computations as efficient as possible, and we'll also need to simplify our notation. There's one insight that we are going to get a lot of mileage out of.

IT'S ALL JUST LINEAR ALGEBRA



VU

When we look at the computation of a neural network, we can see that most operations can be expressed very naturally in those of linear algebra.

The multiplication of weights by their inputs is a multiplication of a matrix of weights by a vector of inputs.

The addition of a bias is the addition of a vector of bias parameters.

The nonlinearities can be implemented as simple element-wise operations.

This perspective buys us two things. First…

$$\mathbf{f}(\mathbf{x}) = \mathbf{V}\,\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

Our notation becomes extremely simple. We can describe the whole operation of a neural network with one easily parseable equation. This expressiveness will be sorely needed when we move to more complicated networks.

MATRIX MULTIPLICATION


```
# k, w, k, h, etc are zero'd arrays
for j in [1 … 2]:
    for i in [1 … 3]:
        k[i] += w[i,j] * x[i]
    k[i] += b[i]

for i in [1 … 3]:
    h[i] = sigmoid(k[i])

for i in [1 … 3]:
    y += h[i] * v[i]
y += c

loss = (y - c) ** 2
```


VU

The second reason is that the biggest computational bottleneck in a neural network is the matrix multiplication. This operation is more than quadratic while everything else is linear.

Matrix multiplication (and other tensor operations like it) can be parallelized and implemented efficiently but it's a lot of work. Ideally, we'd like to let somebody else do all that work (like the implementers of `numpy`) and then just call their code to do the matrix multiplications.

This is especially important if you have access to a GPU. A matrix multiplication can easily be offloaded to the GPU for much faster processing, but for a loop over an array, there's no benefit.

This is called **vectorizing**: taking a piece of code written in for loops, and getting rid of the loops by expressing the function as a sequence of linear algebra operations.

VECTORIZING

Express everything as operations on vectors, matrices and tensors.

Get rid of all the loops

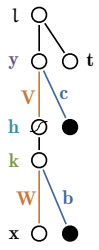
Makes the notation *simpler*.

Makes the execution *faster*.

62

VU

FORWARD



$$l = \sum (y - t)^2$$
$$y = Vh + c$$
$$h = \sigma(k)$$
$$k = Wx + b$$

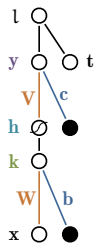
```
k = w.dot(x) + b
h = sigmoid(k)
y = v.dot(h) + c
l = (y - t) ** 2
```

63

VU

Here's what the forward pass looks like in pseudocode. When you implement this in numpy it'll look almost the same.

BUT WHAT ABOUT THE BACKWARD PASS?



$$l = \sum (y - t)^2$$
$$y = Vh + c$$
$$h = \sigma(k)$$
$$k = Wx + b$$

Can we do something like this?

$$\frac{\partial l}{\partial W} \stackrel{?}{=} \frac{\partial l}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial k}\frac{\partial k}{\partial W}$$

64

VU

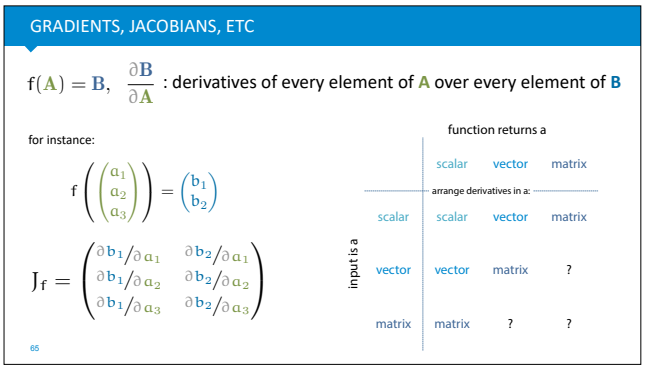
Of course, we lose a lot of the benefit of vectorizing if the backward pass isn't also expressed in terms of matrix multiplications. How do we vectorize the backward pass?

On the left, we see the forward pass of our loss computation as a set of operations on vectors and matrices. (We've generalized things by giving the network a vector output $\mathbf{y}$ and vector target $\mathbf{t}$).

To generalize backpropagation to this view, we might ask if something similar to the chain rule exists for vectors and matrices. Firstly, can we define something analogous to the derivative of one matrix over another, and secondly, can we break this apart into a product of local derivatives,

possibly giving us a sequence of matrix multiplications?

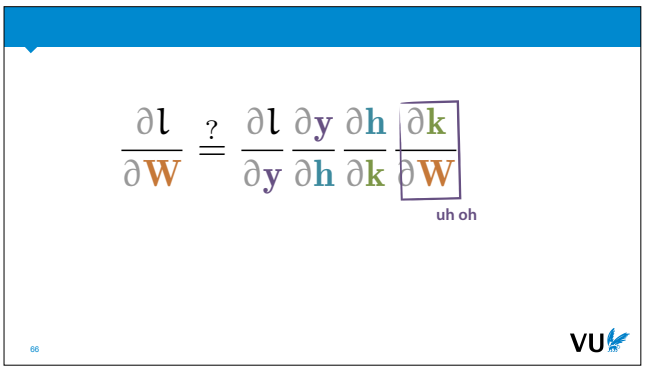
The answer is yes, there are many ways of applying calculus to vectors and matrices, and there are many chain rules available in these domains. However, things can quickly get a little hairy, so we need to tread carefully.



The derivatives of high-dimensional objects are easily defined. We simply take the derivative of every input over every output, and we arrange all possibilities into some logical shape. For instance, if we have a vector-to-vector function, the natural analog of the derivative is a matrix with all the partial derivatives in it.

However, once we get to matrix/vector operations or matrix/matrix operations, the only way to logically arrange every input with every output is a tensor of higher dimension than 2.

NB: We don't normally apply the differential symbol to non-scalars like this. We'll introduce better notation later.



As we see, even if we could come up with this kind of chain rule, one of the local derivatives is now a vector over a matrix. The result could only be represented in a 3-tensor. There are two problems with this:

- If the layer has n inputs and n outputs, we are computing n^3 derivatives, even though we were only ultimately interested in n^2 of them (the derivatives of W). In the scalar algorithm we only ever had two nested loops (an n^2 complexity), and we only ever stored one gradient for one node in the computation graph. Now we suddenly have n^3 complexity and n^3 memory use. We were supposed to make things faster.
- We can easily represent a 3-tensor, but there's no obvious, default way to multiply a 3-tensor with a matrix, or with a vector (in fact there are many different ways). The multiplication of the chain rule becomes very complicated this way.

In short, we need a more careful approach.

SIMPLIFYING ASSUMPTIONS

1) The computation graph *always* has a single, scalar output: l

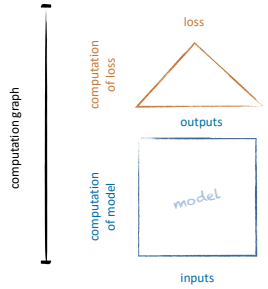
2) We are only ever interested in the derivative of l.

$$\frac{\partial l}{\partial \mathbf{W}}$$ ← scalar ← tensor of any dimension

	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?
3-tensor	3-tensor	?	?

VU

To work out how to do this we make these following simplifying assumptions.



VU

Note that this doesn't mean we can only ever train neural networks with a single scalar output. Our neural networks can have any number of outputs of any shape and size. However, the *loss* we define over those outputs needs to map them all to a single scalar value. The computation graph is always the model, plus the computation of the loss.

THE GRADIENT

We will call $\frac{\partial l}{\partial \mathbf{W}}$ the gradient of l (with respect to $\mathbf{W}$)

Commonly written as $\nabla_{\mathbf{W}} l$

Nonstandard assumption: $\nabla_{\mathbf{W}} l$ has the same *shape* as $\mathbf{W}$

$$[\nabla_{\mathbf{W}} l]_{ijk} = \frac{\partial l}{\partial W_{ijk}}$$

VU

We call this derivative the gradient. This is a common term, but we will deviate from the standard approach in one respect.

Normally, the gradient defined as a row vector, so that it can operate on a space of column vectors by matrix multiplication.

In our case, we are never interested in matrix multiplying the gradient by anything. We only ever want to sum the gradient of l wrt **W** with the original matrix **W** in a gradient update step. For this reason we define the gradient as having the same shape as the tensor with respect to which we are taking the derivative.

In the example shown, **W** is a 3-tensor. The gradient of l wrt **W** has the same shape as **W**, and at element (i, j, k) it holds the scalar derivative of l wrt $\mathbf{W}_{ijk}$.

With these rules, we can use tensors of any shape and dimension and always have a well defined gradient.

NEW NOTATION: THE GRADIENT *FOR* W

$$\nabla_W l = W^\nabla$$

<- always the same

<- actual object of interest

$$b^\nabla = \nabla_b l = \left[\frac{\partial l}{\partial b_1} \dots \frac{\partial l}{\partial b_n}\right]^T$$

$$y^\nabla = \nabla_y l = \frac{\partial l}{\partial y}$$

VU

The standard gradient notation isn't very suitable for our purposes. It puts the loss front and center, but that will be the same for all our gradients. The object that we're actually interested in is relegated to a subscript. Also, it isn't very clear from the notation what the shape is of the tensor that we're looking at.

We can think of the nabla as an operator like a transposition taking an inverse. It turns a matrix into another matrix, a vector into another vector and so on.

For this reason, we'll introduce a new notation. This isn't standard, so don't expect to see it anywhere else, but it will help to clarify our mathematics a lot as we go forward.

We've put the W front and center, so it's clear that the result of taking the gradient is also a matrix, and we've remove the loss, since we've assume that we are always taking the gradient of the loss.

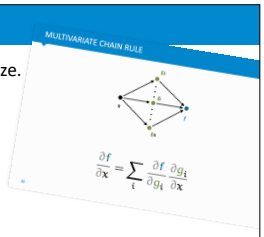
The notation works the same for vectors and even for scalars. When we refer to a single element of the gradient, we will follow our convention for matrices, and use the non-bold version of its letter.

This is the gradient *of* l. Since l never changes, we'll refer to this as "the gradient *for* W".

TENSOR BACKPROPAGATION

Work out *scalar* derivatives first, then vectorize.

Use the **multivariate chain rule** to derive the scalar derivative.



Apply the chain rule step by step.

Start at the loss and work backward, *accumulating* the gradients.

71

VU

This gives us a good way of thinking about the gradients, but we still don't have a chain rule to base out back propagation on.

The main trick we will use is to stick to scalar derivatives as much as possible. Once we have worked out the derivative in purely scalar terms, we will find a way to vectorize their computation.

GRADIENT FOR y

$$y^\nabla$$

$$y_i^\nabla = \frac{\partial l}{\partial y_i} = \frac{\partial \sum_k (y_k - t_k)^2}{\partial y_i} = \sum_k \frac{\partial (y_k - t_k)^2}{\partial y_i} = \sum_k 2(y_k - t_k)\frac{\partial y_k - t_k}{\partial y_i} = 2(y_i - t_i)$$

$$y^\nabla = 2\begin{pmatrix} y_1 - t_1 \\ \vdots \\ y_n - t_n \end{pmatrix} = 2 \cdot (y - t)$$

72

VU

We start simple: what is the gradient for y? This is a vector, because y is a vector. Let's first work out the derivative of the i-th element of y. This is purely a scalar derivative so we can simply use the rules we already know. We get $2(y_i - t_i)$ for that particular derivative.

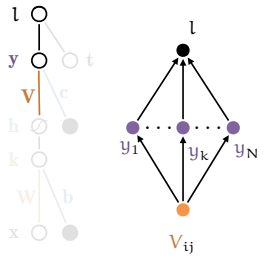
Then, we just re-arrange all the derivatives for y_i into a vector, which gives us the complete gradient for y.

The final step requires a little creativity: we need to figure out how to compute this vector using only basic linear algebra operations on the given vectors and matrices. In this case it's not so complicated: we get the gradient for y by element-

wise subtracting **t** from **y** and multiplying by 2.

We haven't needed any chain rule yet, because our computation graph has only one edge.

GRADIENT FOR V



$$V_{ij}^{\nabla} = \frac{\partial l}{\partial V_{ij}} = \sum_k \frac{\partial l}{\partial y_k}\frac{\partial y_k}{\partial V_{ij}} = \sum_k y_k^{\nabla}\frac{\partial y_k}{\partial V_{ij}} = \sum_k y_k^{\nabla}\frac{\partial [\mathbf{V}\mathbf{h}+\mathbf{c}]_k}{\partial V_{ij}} = \sum_k y_k^{\nabla}\frac{\partial [\mathbf{V}_{k:}\mathbf{h}]_k + c_k}{\partial V_{ij}} = \sum_k y_k^{\nabla}\frac{\partial \sum_l V_{kl}h_l + c_k}{\partial V_{ij}} = \sum_{kl} y_k^{\nabla}\frac{\partial V_{kl}h_l}{\partial V_{ij}} = y_i^{\nabla}\frac{\partial V_{ij}h_j}{\partial V_{ij}} = y_i^{\nabla}h_j$$

$$\mathbf{V}^{\nabla} = \begin{pmatrix} \vdots & \\ \cdots & y_i^{\nabla}h_j \cdots \\ \vdots & \end{pmatrix} = \mathbf{y}^{\nabla}\mathbf{h}^{\mathsf{T}}$$

73

Let's move one step down and work out the gradients for **V**.

RECIPE

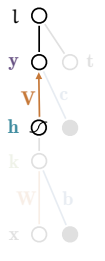
To work out a gradient $\mathbf{X}^{\nabla}$ for some **X**:

- Write down a **scalar derivative** of the loss over *one element* of **X.**
- Use the **multivariate chain** rule to sum over all outputs.
- Work out the scalar derivative.
- **Vectorize** the computation of $\mathbf{X}^{\nabla}$ in terms of the original inputs.

74

VU

GRADIENT FOR h



$$h_i^{\nabla} = \frac{\partial l}{\partial h_i} = \sum_k \frac{\partial l}{\partial y_k}\frac{\partial y_k}{\partial h_i} = \sum_k y_k^{\nabla}\frac{\partial y_k}{\partial h_i} = \sum_k y_k^{\nabla}\frac{\partial [\mathbf{V}\mathbf{h}+\mathbf{c}]_k}{\partial h_i} = \sum_k y_k^{\nabla}\frac{\partial \sum_l V_{kl}h_l + c_k}{\partial h_i} = \sum_{kl} y_k^{\nabla}\frac{\partial V_{kl}h_l + c_k}{\partial h_i} = \sum_k y_k^{\nabla}\frac{\partial V_{ki}h_i}{\partial h_i} = \sum_k y_k^{\nabla}V_{ki}$$

$$\mathbf{h}^{\nabla} = \begin{pmatrix} \vdots \\ \sum_k y_k^{\nabla}V_{ki} \\ \vdots \end{pmatrix} = (\mathbf{y}^{\nabla}\mathbf{1}^{\mathsf{T}} \otimes \mathbf{V})\mathbf{1}$$

or in numpy:

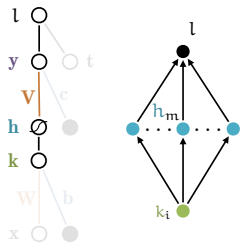
```
(gy[:, None] * v).sum(axis=1)
```

75

Since this is an important principle to grasp, let's keep going until we get to the other set of parameters, **W**. We'll leave the biases as an exercise.

Most of the derivation is the same as before. However, sin

GRADIENT FOR k



$$k_i^\nabla = \frac{\partial l}{\partial k_i}$$
$$= \sum_m h_m^\nabla \frac{\partial h_m}{\partial k_i} = \sum_m h_m^\nabla \frac{\partial \sigma(k_m)}{\partial k_i}$$
$$= h_i^\nabla \frac{\partial \sigma(k_i)}{\partial k_i}$$
$$= h_i^\nabla \sigma'(k_i) = h_i^\nabla \sigma(k_i)(1 - \sigma(k_i))$$
$$= h_i^\nabla h_i(1 - h_i)$$

derivative of sigmoi
σ'(x) = σ(x)(1 - σ(x))

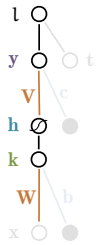
$$\mathbf{k}^\nabla = \mathbf{h}^\nabla \otimes \mathbf{h} \otimes (1 - \mathbf{h})$$

VU

Now, at this point, when we analyze **k**, note that we already have the gradient over **h**. This means that we no longer have to apply the chain rule to anything above **h**. We can draw this scalar computation graph, and work out the local gradient for **k** in terms for the gradient for **h**.

Given that, working out the gradient for **k** is relatively easy, since the operation from **k** to **h** is an element-wise one.

GRADIENT FOR W



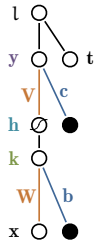
$$W_{ij}^\nabla = \frac{\partial l}{\partial W_{ij}}$$
$$= \sum_m \frac{\partial l}{\partial k_m} \frac{\partial k_m}{\partial W_{ij}} = \sum_m k_m^\nabla \frac{\partial k_m}{\partial W_{ij}}$$
$$= \sum_m k_m^\nabla \frac{\partial [\mathbf{W}\mathbf{x} + \mathbf{b}]_m}{\partial W_{ij}} = \sum_m k_m^\nabla \frac{\partial [\mathbf{W}_{m,}\mathbf{x}]_m + b_m}{\partial W_{ij}}$$
$$= \sum_m k_m^\nabla \frac{\partial \sum_l W_{ml} x_l + b_m}{\partial W_{ij}}$$
$$= \sum_{ml} k_m^\nabla \frac{\partial W_{ml} x_l}{\partial W_{ij}} = k_i^\nabla \frac{\partial W_{ij} x_j}{\partial W_{ij}}$$
$$= k_i^\nabla x_j$$

$$\mathbf{w}^\nabla = \begin{pmatrix} \vdots & \\ \cdots & k_i^\nabla x_j \quad \cdots \\ \vdots & \end{pmatrix} = \mathbf{k}^\nabla \mathbf{x}^T$$

VU

Finally, the gradient for **W**. The situation here is exactly the same as we saw earlier for V (matrix in, vector out, matrix multiplication), so we should expect the derivation to have the same form (and indeed it does).

PSEUDOCODE: BACKWARD




```
y' = 2 * (y - t)

v' = y'.mm(h.T)

h' = y'[None, :] * v).sum(axis=1)

k' = sigmoid(k) * sigmoid(1 - k)

w' = k'.mm(x.T)
```


VU

And here's the backward pass that we just derived.

We've left the derivatives of the bias parameters out. You'll have to work these out to implement the first homework exercise.

(note that ' cannot be part of a variable name in actual python. you'll have to come up with some other name)

RECAP

Vectorizing makes notation simpler, and computation faster.

Vectorizing the **forward pass** is usually easy.

Backward pass: work out the scalar derivatives, accumulate, then vectorize.

VU

Lecture 2: Backpropagation

Peter Bloem
Deep Learning 2020

dlvu.github.io

VU VRIJE UNIVERSITEIT AMSTERDAM

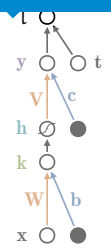
PART FOUR: **AUTOMATIC DIFFERENTIATION**

Letting the computer do all the work.

VU

We've simplified the computation of derivatives a lot. All we've had to do is work out local derivatives and chain them together. We can go one step further: if we let the computer keep track of our computation, and provide some backwards functions for basic operations, the computer can work out the whole backpropagation algorithm for us. This is called **automatic differentiation** (or sometimes **autograd**).

THE STORY SO FAR



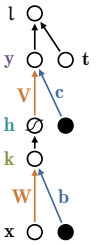
pen and paper

```
v' = y'.mm(h.T)

h' = y'[None, :] * v).sum(axis=1)

k' = sigmoid(k) * sigmoid(1 - k)

w' = k'.mm(x.T)
```

in the computer

VU

THE IDEAL



pen and paper

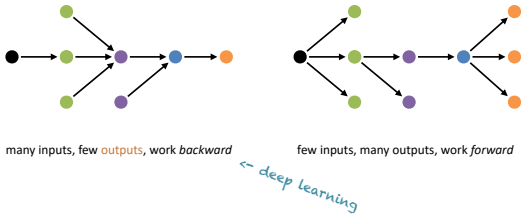
```
k = w.dot(x) + b

h = sigmoid(k)

y = v.dot(h) + c

l = (y - t) ** 2

l.backward() # start backprop
```

in the computer

VU

This is what we want to achieve. We work out on pen and paper the local derivatives of various modules, and then we chain these modules together in a copmputation graph *in our code*. The compute keep the graph in memory, and can automatically work out the backropagation.

AUTOMATIC DIFFERENTIATION



This kind of algorithm is called automatic differentiation. What we've been doing so far is called **backward, or *reverse mode* automatic differentiation**. This is efficient if you have few output nodes. Note that if we had two output nodes, we'd need to do a separate backward pass for each.

If you have few inputs, it's more efficient to start with the inputs and apply the chain rule working forward. This is called forward mode automatic differentiation.

Since we assume we only have one output node (where the gradient computation is concernced), we will always use reverse mode.

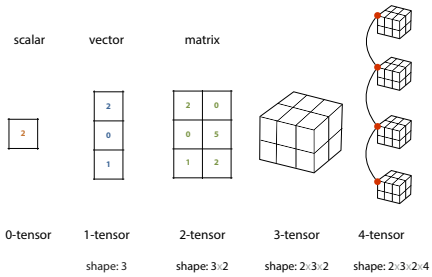
THE PLAN

Tensors

Building computation graphs

Working out backward functions

TENSORS (AKA MULTIDIMENSIONAL ARRAYS)

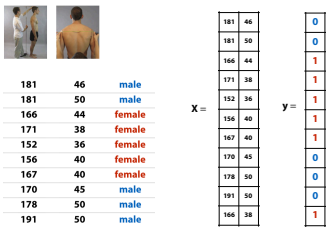


The basic datastructure of our system will be the tensor. A tensor is a generic name for family of datastructures that includes a scalar, a vectorm, a matrix and so on.

There is no good way to visualize a 4-dimensional structure. We will occasionally use this form to indicate that there is a fourth dimension along which we can index the tensor.

We will assume that whatever data we work with (images, text, sounds), will in some way be encode into one or more tensors, before it is fed into our system. Let's look at some examples.

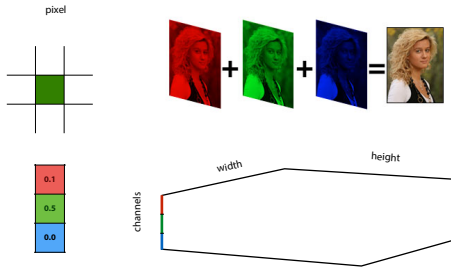
A CLASSIFICATION TASK AS TWO TENSORS



A simple dataset, with numeric features can simply be represented as a matrix. For the labels, we usually create a separate corresponding vector for the labels.

Any categoric features or labels should be converted to numeric features (normally by one-hot coding).

AN IMAGE AS A 3-TENSOR



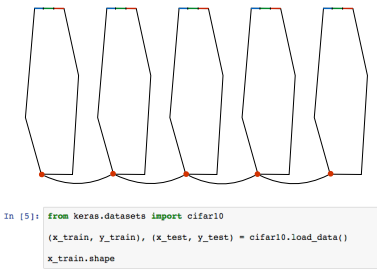
88

Images can be represented as 3-tensors. In an RGB image, the color of a single pixel is represented using three values between 0 and 1 (how red it is, how green it is and how blue it is). This means that an RGB image can be thought of as a stack of three color channels, represented by matrices.

This stack forms a 3-tensor.

source: **http://www.adsell.com/scanning101.html**

A DATASET OF IMAGES



```
In [5]: from keras.datasets import cifar10
        (x_train, y_train), (x_test, y_test) = cifar10.load_data()
        x_train.shape
Out[5]: (50000, 32, 32, 3)
```

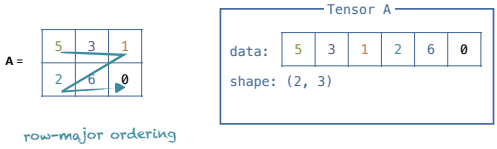
89

If we have a dataset of images, we can represent this as a 4-tensor, with dimensions indexing the instances, their width, their height and their color channels respectively. Below is a snippet of code showing that when you load the CIFAR10 image data in Keras, you do indeed get a 4-tensor like this.

There is no agreed standard ordering for the dimensions in a batch of images. Tensorfflow and Keras use (batch, height, width, channels), whereas Pytorch uses (batch, channels, height, width).

(You can remember the latter with the mnemonic "**bach**elor **chow**".)

TENSORS IN MEMORY: ROW MAJOR ORDERING



90

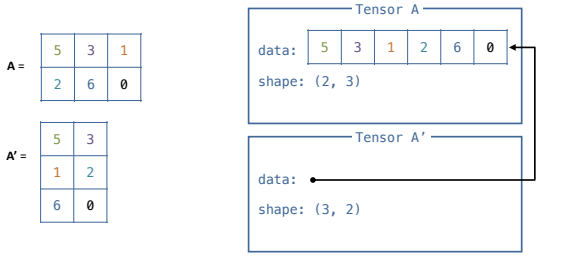
It's important to realize that even though we think of tensors as multidimensional arrays, in memory, they are necessarily laid out as a single line of numbers. The Tensor object knows the shape that these numbers should take, and uses that to compute whatever we need it to compute.

We can do this in two ways: scanning along the rows first and then the columns is called **row-major ordering**. This is what numpy and pytorch both do. The other option is (unsurprisingly) called column major ordering. In higher dimensional tensors, the dimensions further to the right are always scanned before the dimensions to their left.

Imagine looping over all elements in a tensor with shape (a, b, c, d) in four nested loops. If you want to loop over the elements in the order they are in memory, then the loop over d would be the innermost loop, then c, the b and then a.

This allows us to perform some operations very cheaply, **but we have to be careful**.

RESHAPE/VIEW



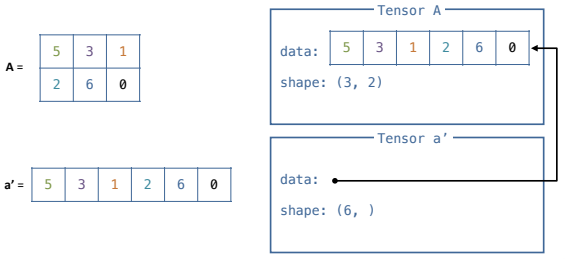
91

VU

Here is one such cheap operation: a reshape. This changes the shape of the tensor, but not the data.

To do this cheaply, we can create a new tensor object with the same data as the old one, but with a different shape.

RESHAPE/VIEW



92

VU

EXAMPLE: NORMALIZE COLORS

```
images = load_images(…)
n, h, w, c = images.size()

images = images.reshape(n*h*w, c)
images = images / images.max(dim=0)
images = images.reshape(n, h, w, c)

# this makes no sense, but it is allowed.
images = images.reshape(n*h, w*c)
images = images.reshape(h, n, c, w)
```

93

VU

Imagine that we wanted to scale a dataset of images in such way that over the whole dataset, each color channel has a maximal value of 1 (independent of the other channels).

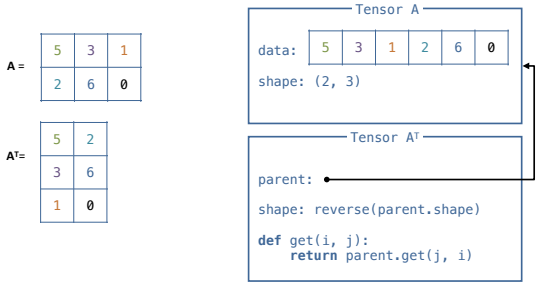
To do this, we need a 3-vector of the maxima per color channel. Pytorch only allows us to take the maximum in one direction, but we can reshape the array so that all the directions we're not interested in are collapsed into one dimension.

Afterwards, we can reverse the reshape to get our image dataset back.

We have to be careful: the last three lines in this slide form a perfectly valid reshape, but the `c` dimension in the result does *not* contain our color values.

In general, you're fine if you **collapse dimensions that are next to each other**, and uncollapse them *in the same order.*

TRANSPOSE

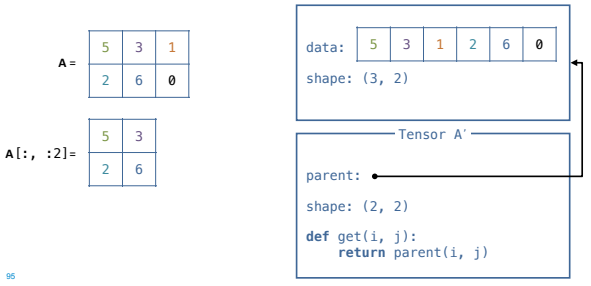


94

A transpose operation can also be achieved cheaply. In this case, we just keep a reference to the old tensor, and whenever a user requests the element (i, j) we return (j, i) from the original tensor.

NB: This isn't precisely how pytorch does this, but the effect is the same: we get a transposed tensor in constant time, by viewing the same data in memory in a different way.

SLICE



95

Even slicing can be accomplished by referring to the original data.

NON-CONTIGUOUS

Contiguous tensor: the data are directly laid out in row-major ordering for the shape with no gaps or shuffling required.

```
x = torch.randn(2, 3)
x = x[:, 1:]
x.view(6)
Traceback (most recent call last):
…
RuntimeError: view size is not compatible with input tensor's size and
stride (at least one dimension spans across two contiguous subspaces).
Use .reshape(...) instead.
```

`x.contiguous()`: make contiguous (by copying the data).

`x.view()`: reshape if possible without making contiguous.

`x.reshape()`: reshape, call `contiguous()` if necessary.

96

VU

For some operations, however, the data needs to be contiguous. That is, the tensor data in memory needs to be one uninterrupted string of data in row major ordering with no gaps. If this isn't the case, pytorch will throw an exception like this.

You can fix this by calling `.contiguous()` on the tensor. The price you pay is the linear time and space complexity of copying the data.

Tensors

Building computation graphs

Working out backward functions

97

VU

COMPUTATION GRAPHS

bipartite: only op-to-tensor or tensor-to-op edges

tensor nodes: no input or one input, multiple outputs

operation nodes: multiple inputs and outputs

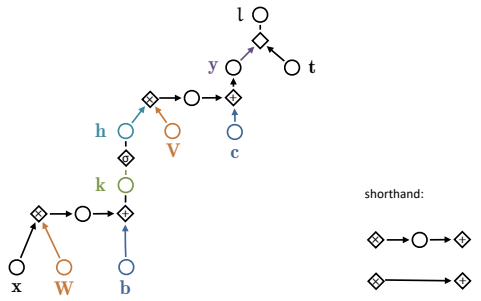
VU

We'll need to be a little mode precise in our notations. From now on we'll draw computation graphs like this, making the operation explicit.

(There doesn't seem to be a standard notation. This will work for our purposes).

In tensorflow operations are called *ops*, and in pytorch they're called *functions*.

FEEDFORWARD NETWORK



VU

As an example, here is our MLP expressed in our new notation.

Just as before, it's up to us how *granular* we make the computation. We can wrap the whole computation of the loss in a single operation, but we can also separate the matrix multiplication by the weights and the addition of the bias vector into separate operations.

Note that the *granularity* with which we represent a computation in a computation graph is up to us. We could, for instance also choose to combine the multiplication and the addition into a single operation.

If there's a uniform direction to the computation, we'll leave out the arrowheads for clarity, but you should think of a computation graph as a directional one. All connections have a definite direction.

When we draw intricate computation graphs, we may connect an op node to another opnode, without explicitly drawing the tensor node in between. This should be understood as a shorthand for a proper bipartit

---

RESTATING OUR ASSUMPTIONS

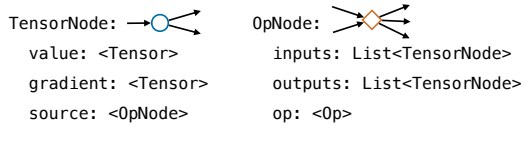
One output l node with a scalar value.

All gradients are of l, with respect to the tensors on the tensor nodes.

VU

We hold on to the same assumptions we had before. Without these two assumptions things would become a lot more complicated.

IMPLEMENTATIONS



To store a computation graph in memory, we need three objects.

A TensorNode objects holds the tensorvalue at that node. It holds the gradient of the tensor at that node (to be filled by the backpropagation algorithm) and it holds a pointer to the Operation Node that produced it (which can be null for leaf nodes).

An OpNode object represents an instance of a particular

DEFINING AN OPERATION

```
Op:
  def forward(context, inputs):
    # given the inputs, compute the outputs
    ...

  def backward(context, outputs_gradient):
    # given the gradient of the loss wrt to the ouputs
    # compute the gradient of the loss wrt to the inputs
    ...
```

stores anything we need for the backward pass

$$f(\mathbf{A}) = \mathbf{B}$$

$$\text{forward}_f : \mathbf{A} \mapsto \mathbf{B}$$

$$\text{backward}_f : \mathbf{B}^{\nabla} \mapsto \mathbf{A}^{\nabla}$$

We also need to implement the computation of the operation itself. This is done in two functions (these are usually class functions or static functions).

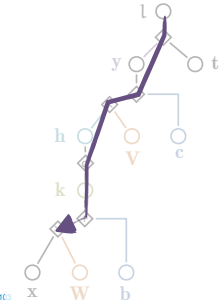
The function forward computes the outputs given the inputs (just like any function in any programming language).

The function backward takes the gradient for the outputs (the gradient of the loss wrt to the outputs) and produces the gradient for the inputs.

Both functions are also given a `context` object. This is a datastructure (a dictionary or a list) to which the forward can add any value which it needs to save for the backward.

Note that the backward function does not compute the local derivative: it computes the accumulated gradient of the loss over the inputs (given the accumulated gradient of the loss over the outputs).

BACKPROPAGATION



build a computation graph, perform forward pass, bottom-up from leaves.

traverse the tree backwards breadth-first

call backward for each OpNode

breadth-first ensures that the output gradients are known

add the computed gradients to the tensornodes

sum them with any gradients already present

This is the algorithm for backpropagation in this setting. We chain together these modules into a computation graph, and perform a forward pass to compute a loss. We then walk backward from the loss node breadth-first to compute the gradients for each tensor node. Because we walk breadth first, we ensure that we've always computed the gradients for the outputs of each Operation node we encounter already, and we can simply call its backward to compute the gradients for its inputs.

Note the last point: some tensor nodes will be inputs to multiple operation nodes. By the multivariate chain rule, we should sum the gradients they get from all the OpNodes they feed into. We can achieve this easily by just initializing

the gradients to zero and adding the gradients we compute to any that are already there.

BUILDING COMPUTATION GRAPHS IN CODE

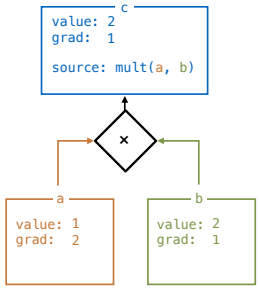
Lazy execution: build your graph, compile it, feed data through.

Eager execution: perform forward pass, keep track of computation graph.

There are two common strategies for constructing the computation graph: lazy and eager execution.

EXAMPLE: LAZY EXECUTION

```
a = TensorNode()
b = TensorNode()

c = a * b

m = Model(
  in=(a, b),
  loss=c)

m.train((1, 2))
```



Here's one example of how a lazy execution API might look.

Note that when we're building the graph, we're not telling it which values to put at each node. We're just defining the *shape* of the computation, but not performing the computation itself.

When we create the model, we define which nodes are the input nodes, and which node is the loss node. We then provide the input values and perform the forward and backward passes.

BUILDING THE COMPUTATION GRAPH: LAZY EXECUTION

Tensorflow 1.x default, Keras default

Define the computation graph.
- Compile it.
- Iterate backward/forward over the data

Fast. Many possibilities for optimization. Easy to serialise models. Easy to make training parallel.

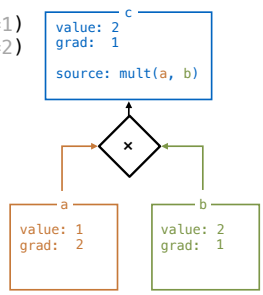
Difficult to debug. Model must remain static during training.

In lazy execution, which was the norm during the early years of deep learning, we build the computation graph, but we don't yet specify the values on the tensor nodes. When the computation graph is finished, we define the data, and we feed it through.

This is fast since the deep lerning system can optimize the graph sturcture during compilation, but it makes models hard to debug: if something goes wrong during training, you probably made a mistake while defining you graph, but you will only get an while passing data through it. The resulting stack trace will never point to the part of your code where you made the mistake.

EXAMPLE: EAGER EXECUTION

```
a = TensorNode(value=1)
b = TensorNode(value=2)

c = a * b

c.backward()
```



VU

In eager mode deep learning systems, we create a node in our computation graph (a TensorNode) by specifying what data it should contain. The result is a tensor object that stores both the data, and the gradient over that data (which will be filled later).

Here we create the variables a and b. If we now apply an operation to these, for instance to multiply their values, the result is another variable c. Languages like python allow us to *overload* the * operator it looks like we're just computing multiplication, but behind the scenes, we are creating a computation graph that records all the computations we've done.

We compute the data stored in c by running the computation, bu we also store references to the variables that were used to create c, and the operation that created it.

Using this graph, we can perform the back propagation from a given node that we designate as the loss node. We work our way down the graph computing the derivative of each variable with respect to c. At the start the TensorNodes do not have their grad's filled in, but at the end of the backward, all gradients have been computed.

BUILDING THE COMPUTATION GRAPH: EAGER EXECUTION

PyTorch, Tensorflow 2.0 default, Keras option

- Build the computation graph on the fly during the forward pass.

Easy to debug, problems in the model occur as the module is executing.
Flexible: the model can be entirely different from one forward to the next.

More difficult to optimize. A little more difficult to serialize.

VU

In eager execution, we simply execute all operations immediately on the data, and collect the computation graph on the fly. We then execute the backward and ditch the graph we've collected.

This makes debugging much easier, and allows for more flexibility in what we can do in the forward pass. It can, however be a little difficult to wrap your head around. Since we'll be using pytorch later in the course, we'll show you how eager execution works step by step.

~~Tensors~~

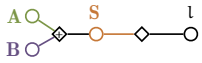
~~Building computation graphs~~

Working out backward functions

VU

WORKING OUT THE BACKWARD FUNCTION

```
class Plus(Op):

  def forward(context, a, b):

    # a, b are matrices of the same size

    return a + b


  def backward(context, goutput):

    return goutput, goutput
```



$$A^{\nabla}_{ij} = \frac{\partial l}{\partial A_{ij}}$$
$$= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial A_{ij}} = \sum_{kl} S^{\nabla}_{kl} \frac{\partial S_{kl}}{\partial A_{ij}}$$
$$= \sum_{kl} S^{\nabla}_{kl} \frac{\partial [A + B]_{kl}}{\partial A_{ij}} = \sum_{kl} S^{\nabla}_{kl} \frac{\partial A_{kl} + B_{kl}}{\partial A_{ij}}$$
$$= S^{\nabla}_{ij} \frac{\partial A_{ij}}{\partial A_{ij}} = S^{\nabla}_{ij}$$

$$A^{\nabla} = S^{\nabla} \qquad B^{\nabla} = S^{\nabla}$$

VU

The final ingredient we need is a large collection of operations with backward functions worked out. We'll show how to do this for a few examples.

First, an operation that sums two matrices element-wise.

The recipe is the same as we saw in the last part:

1) Write out the scalar derivative for a single element.
2) Use the multivariate chain rule to sum over all outputs.
3) Vectorize the result.

Note that when we drawe the computation graph, we can think of everything that happens between S and l as a single module: we are already given the gradient of l over S, so it doesn't matter if it's one operation or a million.

Again, we can draw the computation graph at any granularity we like: very fine individual operations like summing and multiplying or very coarse-grained operations like entire NNs.

For this operation, the context object is not needed, we can perform the backward pass without remembering anything about the forward pass.

BACKWARD: A FEW MORE EXAMPLES

Sigmoid

Row-wise sum

Expand

VU

To finish up, we'll show you the implementation of some more operations. You'll be asked to do a few of these in the second homework exercise.

SIGMOID

```
class Sigmoid(Op):

  def forward(context, x):
    # x is a tensor of any shape
    sigx = 1 / (1 + (- x).exp())
    context['sigx'] = sigx
    return sigx


  def backward(context, goutput):
    sigx = context['sigx']
    return goutput * sigx * (1 - sigx)
```

$$X^{\nabla}_{ijk} = \sum_{abc} Y^{\nabla}_{abc} \frac{\partial Y_{abc}}{\partial X_{ijk}}$$
$$= \sum_{abc} Y^{\nabla}_{abc} \frac{\partial \sigma(X_{abc})}{\partial X_{ijk}}$$
$$= Y^{\nabla}_{ijk} \frac{\partial \sigma(X_{ijk})}{\partial X_{ijk}}$$
$$= Y^{\nabla}_{ijk} \sigma(X_{ijk}) (1 - \sigma(X_{ijk}))$$

$$X^{\nabla} = Y^{\nabla} \otimes \sigma(X) \otimes (1 - \sigma(X))$$

This is a pretty simple derivation, but it shows two things:

1) We can easily do a backward over functions that output high dimensional tensors, but we should sum over all dimensions of the output when we apply the multivariate chain rule.
2) The backward function illustrates the utility of **the context object**. To work out the backward, we need to know the original input. We could just have stored that for every operation, but as we see here, we've done part of the computation already (an in other backwards the inputs *aren't* necessary). If we give the operation control over what information it

wants to store for the backward pass, we are maximally flexible.

ROW-WISE SUM

```
class RowSum(Op):

  def forward(context, x):
    # x is a matrix
    sumd = x.sum(axis=1)
    context['m'] = x.shape[1]
    return sumd

  def backward(context, gy):
    n, m = gy.shape[0], context['m']
    return gy[None, :].expand(n, m)
```

$$X^{\nabla}_{ij} = \sum_k y^{\nabla}_k \frac{\partial y_k}{\partial X_{ij}} = \sum_k y^{\nabla}_k \frac{\partial \sum_l X_{kl}}{\partial X_{ij}}$$
$$= \sum_{kl} y^{\nabla}_k \frac{\partial X_{kl}}{\partial X_{ij}} = y^{\nabla}_i \frac{\partial X_{ij}}{\partial X_{ij}}$$
$$= y^{\nabla}_i$$

$$\mathbf{X}^{\nabla} = \mathbf{y}^{\nabla}\mathbf{1}^{\mathsf{T}}$$

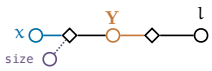
Note that the output is a vector, because we've summed out one dimension. Therefore, when we apply the multivariate chain rule, we sum over only one dimension.

This is a sum like the earlier example, so we don't need to save any tensor values from the forward pass. However, we do need to remember the size of the dimension we summed out. Therefore, we use the context to store just one object.

The `expand` function we use here is not available in numpy, but it does exist in pytorch. In numpy we can use `repeat`.

EXPAND

```
class Expand(Op):

  def forward(context, x, size):
    # x is a scalar
    return np.full(x, size=size)

  def backward(context, gy):
    return gy.sum(), None
```



$$x^{\nabla} = \sum_{ab} Y^{\nabla}_{ab} \frac{\partial Y_{ab}}{\partial x}$$
$$= \sum_{ab} Y^{\nabla}_{ab} \frac{\partial x}{\partial x}$$
$$= \sum_{ab} Y^{\nabla}_{ab}$$

Our final example includes a constant argument. We take a scalar, and expand it to a matrix of the given size, filled with the value x.

In this case, we do not care about the gradient for size (and even if we did, integer values don't yield meaningful gradients without some extra work). We consider this a constant.

This can be done in different ways. In pytorch, you return None for the gradient for a constant (as we've shown here).

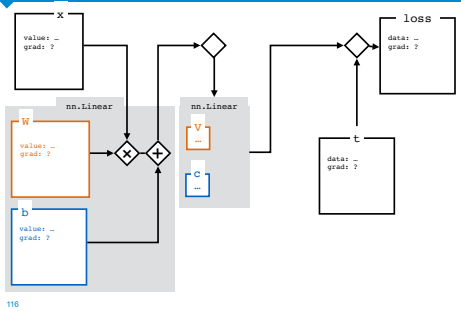
In our toy system (vugrad), we consider all *keyword arguments* constants.

~~Tensors~~

~~Building computation graphs~~

~~Working out backward functions~~

MODULES OR LAYERS



VU

Most deep learning frameworks also have a way of combining model parameters and computation into a single unit, often called a **module** or a **layer**.

In this case a Linear module (as it is called in Pytorch) takes care of implementing the computation of a single layer of a neural network (sans activation) and of remembering the weights and the bias.

Modules have a ***forward*** function which performs the computation of the layer, but they *don't* have a **backward** function. They just chain together operations, and the backpropagation calls the backwards on the operations. In other words, it doesn't matter to the backpropagation whether we use a module or apply all the operations by hand.

THE IDEAL

pen and paper
work out forward and backward functions


```
k = w.dot(x) + b
h = sigmoid(k)
y = v.dot(h) + c
l = (y - t) ** 2
l.backward() # start backprop
```

compute forward pass and build computation graph

in the computer

VU

Whichcompletes the picture we wanted to create: a system where all we have to do is perform some computations on some tensors, just like we would do in `numpy`, and all the building of computation graphs and all the backpropagating is handled for us automatically.

LECTURE RECAP

neural networks: historically inspired by brain structure, but really just a pile of differentiable linear algebra.

backpropagation: work out local gradients symbolically, global gradients numerically. Work from back to front, applying the chain rule.

tensor backpropagation: assume scalar loss `l`, always compute the gradient of the loss.

automatic differentiation: Build computation graph on the fly. Backpropagation as a *breadth-first* walk down from the loss.

VU

So

THANK YOU FOR YOUR ATTENTION

dlvu@peterbloem.nl