NEUR 503

Credit assignment

Blake Richards

Mila/McGill/CIFAR

Outline

1. Basics of credit assignment
2. The credit assignment problem
3. The backpropagation-of-error algorithm
4. Feedback alignment
5. Kolen-Pollack

Basics of credit assignment

What is credit assignment?

When we refer to ***credit assignment***, what are we talking about?

To answer this question, we first have to answer another...

What is learning?

Answer: Learning is a process of change in the brain that leads us to get ***better*** at something.

What is credit assignment?

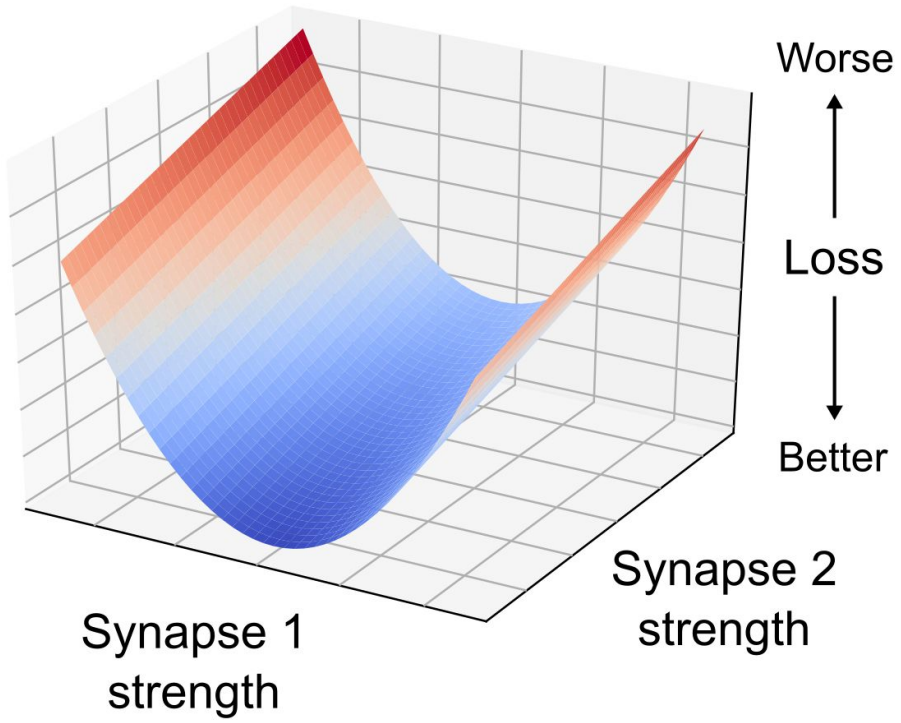
Credit assignment is the process of determining what needs to change in order to get better.

In other words, we assign credit to different parts of the brain for our errors or successes, and try to change them in order to reduce errors and increase successes.

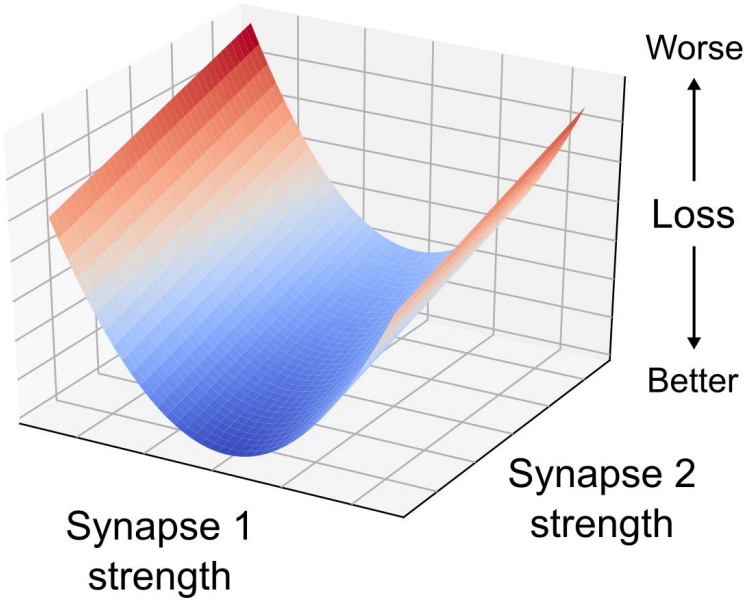
If we assume that the changes that underlie learning are changes in *synaptic strengths*, then credit assignment is the process of figuring out which synapses need to change (and in which way) in order to get better.

Quantifying better

We can quantify what it means to be “better” using a loss function (sometimes also called a cost function).



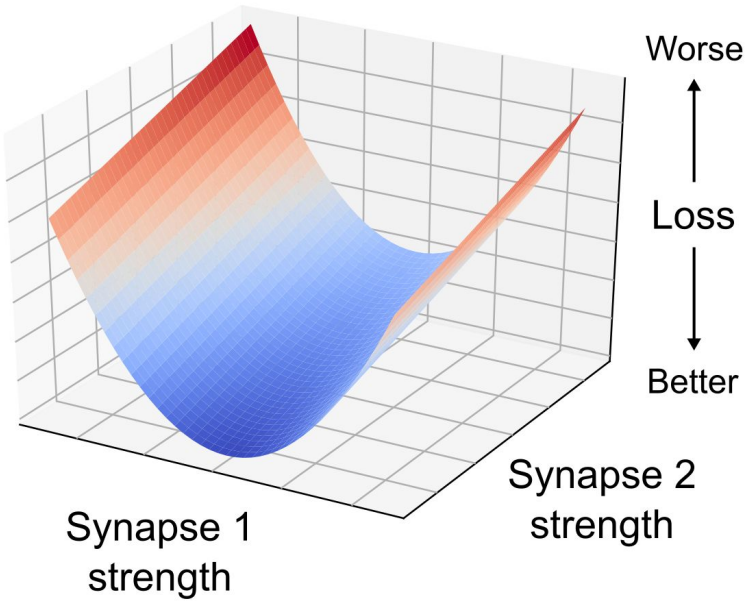
Quantifying learning with loss functions



$$\mathcal{L}(\mathbf{W}; D) : \mathbb{R}^n \times R^K \to \mathbb{R}$$

Where $\mathbf{W} \in \mathbb{R}^n$ are the current synaptic weights, and $D = \{D_1, D_2, \ldots, D_K\}$ are a set of data points.

Quantifying learning with loss functions



Our goal in learning is to find a set of weights, $\mathbf{W}$, that minimizes $\mathcal{L}(\mathbf{W}; D)$.

How can we reduce our loss function? Perturbation methods.

Make random changes and keep those that improve things:

Algorithm 1: Weight perturbation for loss function reduction

Result: $\text{argmin}_{\mathbf{W}} \mathcal{L}(\mathbf{W}; D)$

$\Delta\mathcal{L} \leftarrow \infty$;

while $\underline{|\Delta\mathcal{L}| > \theta}$ **do**

  $\Delta\mathbf{W} \sim \mathcal{N}(0; \sigma^2)$;

  $\mathbf{W}' \leftarrow \mathbf{W} + \eta\Delta\mathbf{W}$;

  $\Delta\mathcal{L} = \mathcal{L}(\mathbf{W}; D) - \mathcal{L}(\mathbf{W}'; D)$;

  $\mathbf{W} \leftarrow \mathbf{W} + \eta\Delta\mathcal{L}\Delta\mathbf{W}$;

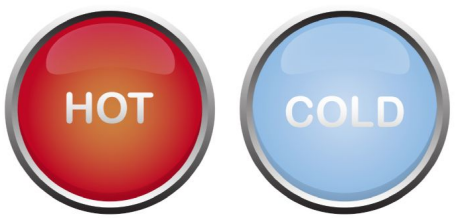
end

Where η is a "learning rate", θ is a threshold to determine when to stop, and σ^2 is the variance in our synaptic noise.

How can we reduce our loss function? Perturbation methods.

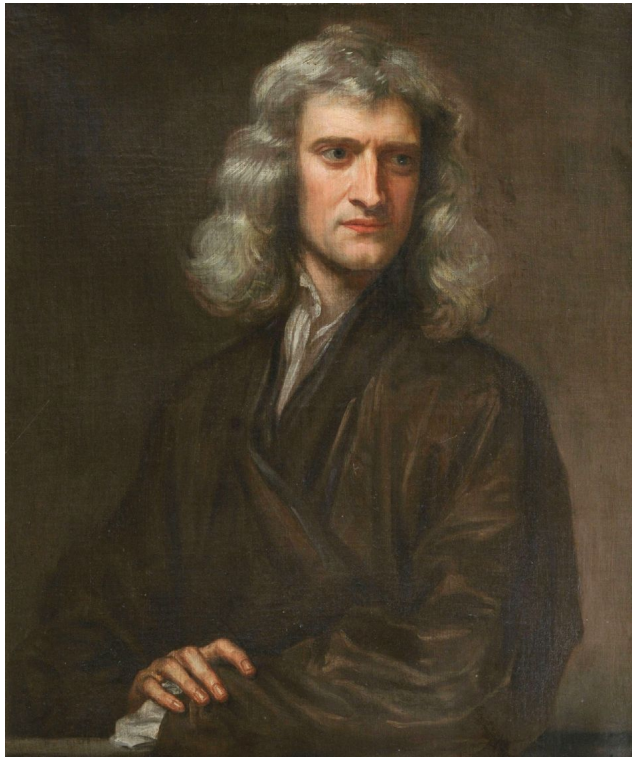
The problem with perturbation strategies is that they scale poorly. As you increase the number of dimensions (e.g. the number of synaptic weights) the time it takes to find the minimum increases.

Think for example, about playing “hot or cold” in one million dimensional space! (We’ll come back to this...)



How can we reduce our loss function? Newton's method.

If we can calculate **gradients** and **Hessians** on our loss function, then Newton had a better approach.



Reminder, what are gradients and Hessians?

Gradients and Hessians are just high-dimensional generalizations of derivatives and second derivatives.

For example, if we have a function $f\colon \mathbb{R}^n \to \mathbb{R}$, on inputs $\mathbf{x} = \{x_1, \ldots, x_n\}$ then our **gradient** $(\nabla_{\mathbf{x}} f(\mathbf{x}))$ is defined as:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Reminder, what are gradients and Hessians?

And our **Hessian** ($\nabla^2_{\mathbf{x}} f(\mathbf{x})$) is defined as:

$$\nabla^2_{\mathbf{x}} f(\mathbf{x}) = 0 \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial^2 x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial^2 x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial^2 x_n} \end{bmatrix}$$

How can we reduce our loss function? Newton's method.

Algorithm 2: Newton's method for function minimization

Result: $\text{argmin}_{\mathbf{W}} \mathcal{L}(\mathbf{W}; D)$

$\Delta\mathcal{L} \leftarrow \infty$;

while $\underline{|\Delta\mathcal{L}| > \theta}$ **do**

$\Delta\mathbf{W} \leftarrow -\eta(\nabla^2_{\mathbf{W}}\mathcal{L}(\mathbf{W}; \mathcal{D}))^{-1}\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}; \mathcal{D})$;

$\mathbf{W}' \leftarrow \mathbf{W} + \Delta\mathbf{W}$;

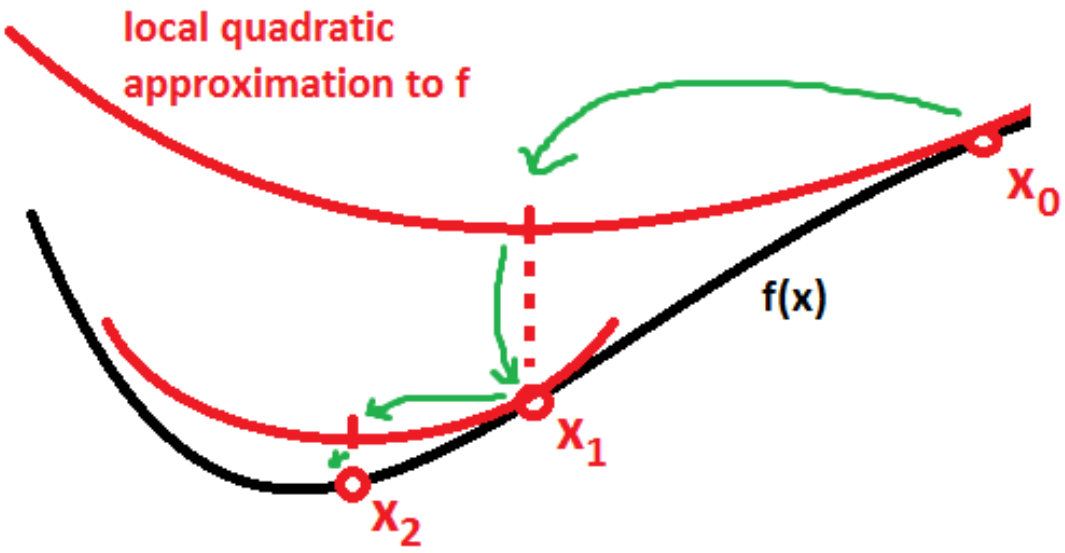
$\Delta\mathcal{L} = \mathcal{L}(\mathbf{W}; D) - \mathcal{L}(\mathbf{W}'; D)$;

$\mathbf{W} \leftarrow \mathbf{W}'$;

end

How can we reduce our loss function? Newton's method.

One way of understanding Newton's method is that it is doing successive quadratic approximations to minimize the function.



How can we reduce our loss function? Newton's method.

The problem with Newton's method is that it is impractical to calculate the inverse of the Hessian ($(\nabla^2_{\mathbf{W}}\mathcal{L}(\mathbf{W};\mathcal{D}))^{-1}$) in high-dimensional systems.

So, we need something like this, but simpler... (sorry Newton)

How can we reduce our loss function? Gradient descent.

We can simplify Newton's method by just using the first-order information, this is known as **gradient descent** (because you're going down the gradient).

Algorithm 3: Gradient descent for loss function reduction

Result: $\text{argmin}_{\mathbf{W}} \mathcal{L}(\mathbf{W}; D)$

$\Delta\mathcal{L} \leftarrow \infty$;

while $\underline{|\Delta\mathcal{L}| > \theta}$ **do**

$\Delta\mathbf{W} \leftarrow -\eta\nabla_{\mathbf{W}}\mathcal{L}(\mathbf{W}; \mathcal{D})$;

$\mathbf{W}' \leftarrow \mathbf{W} + \Delta\mathbf{W}$;

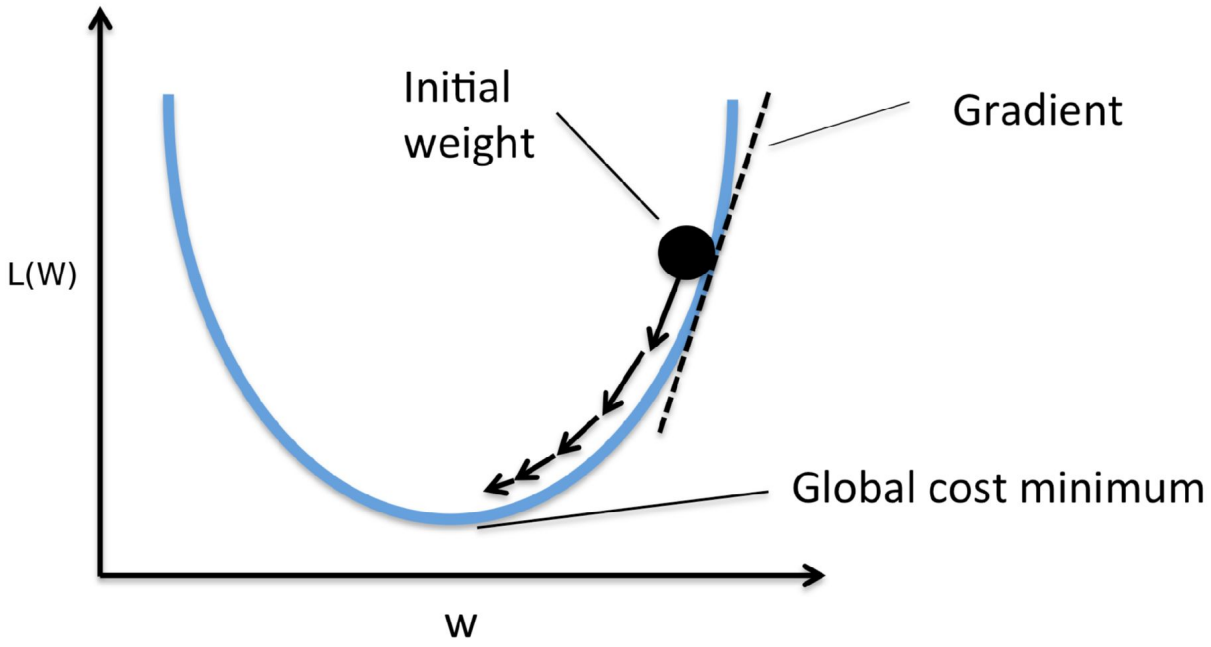
$\Delta\mathcal{L} = \mathcal{L}(\mathbf{W}; D) - \mathcal{L}(\mathbf{W}'; D)$;

$\mathbf{W} \leftarrow \mathbf{W}'$;

end

How can we reduce our loss function? Gradient descent.

Gradient descent is essentially a linear version of Newton's method where we use a first-order Taylor series rather than a second-order one.



Which is better, gradient descent or perturbation methods?

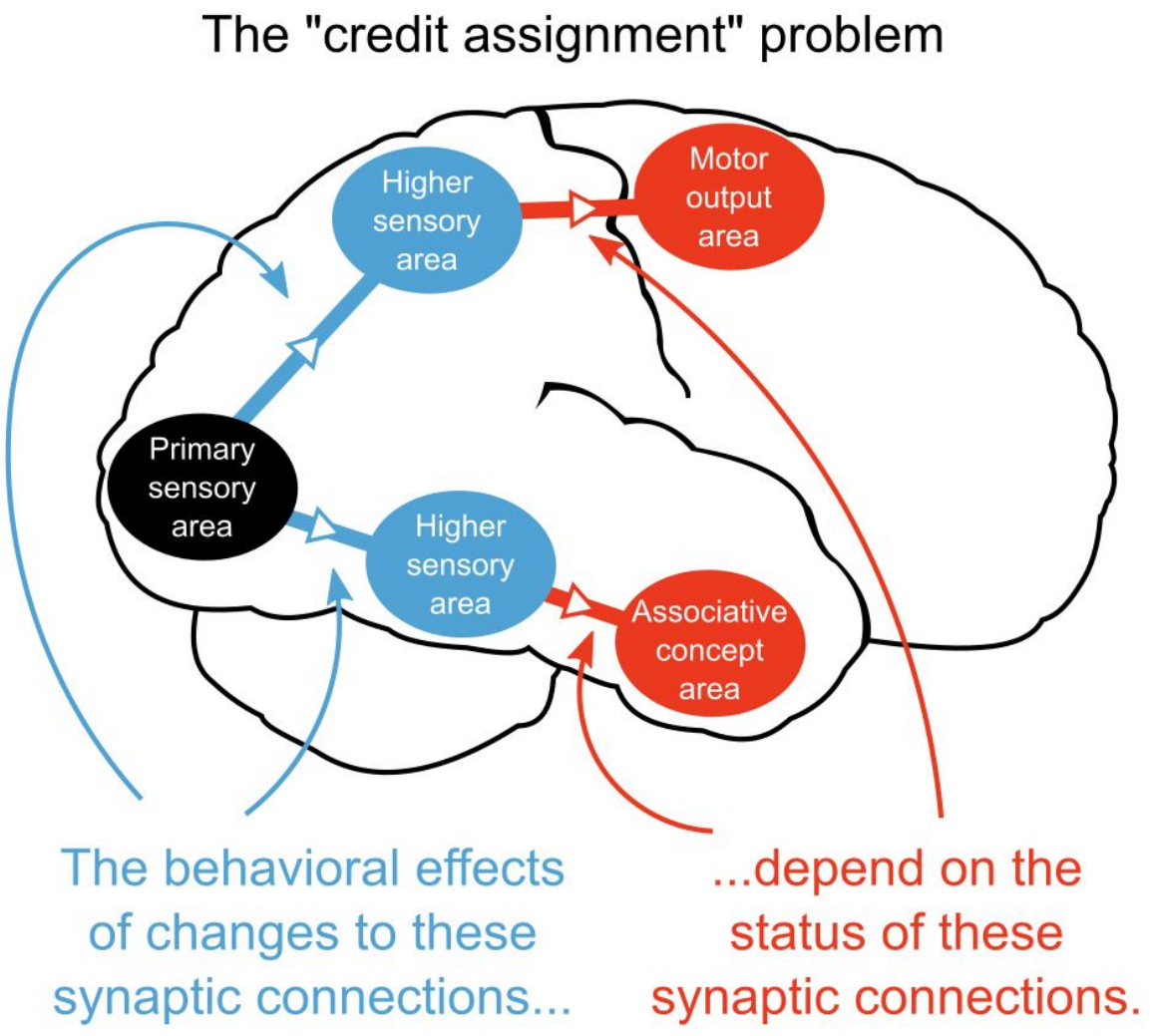
Perturbation methods seem like a nice approach, they're intuitive and you don't even need to calculate a gradient.

The problem is, perturbation methods are ***slow***, because they are a **zeroth-order method** (in contrast to gradient descent, which is first-order, and Newton's method, which is second-order).

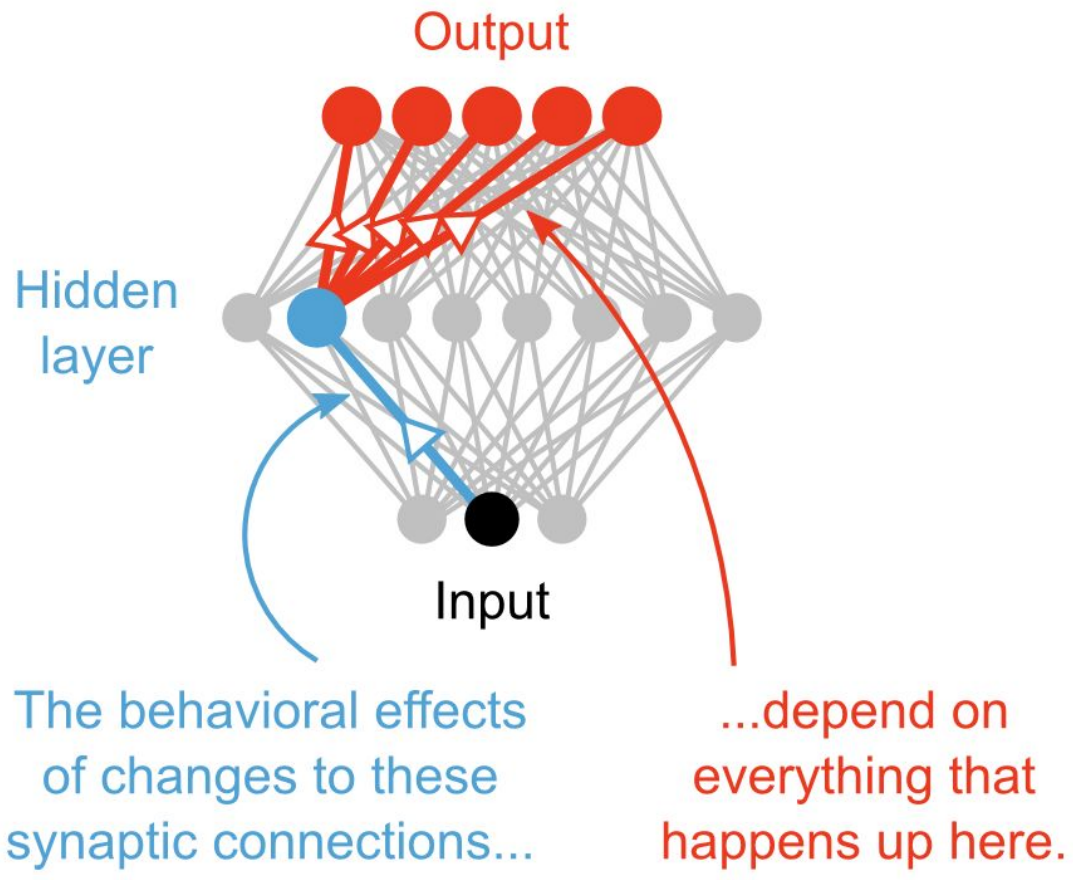
You will see how slow perturbation methods are in the assignment!

The credit assignment problem

The credit assignment problem



The credit assignment problem



The backpropagation-of-error algorithm

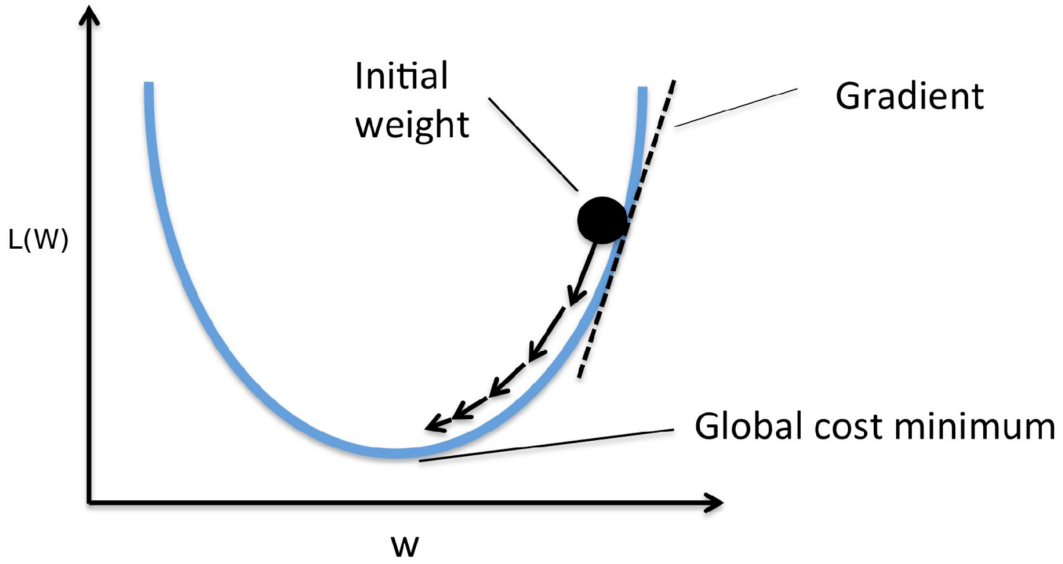
Backpropagation-of-error

Another way of phrasing the credit assignment problem is: if you have internal representations in a network, how can you learn those representations?

The standard algorithm for this is the **backprogagation-of-error algorithm**, which Rumelhart, Hinton & Williams (1986) popularized. (Note: others had invented it before them.)

Backpropagation-of-error

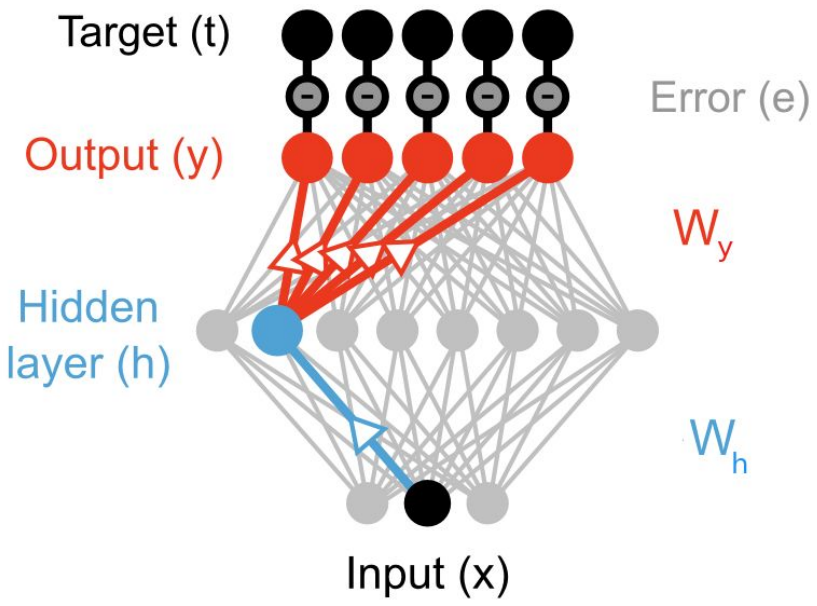
Backprop, as it is often called, is really just an implementation of gradient descent!



Chain rule to the rescue

To calculate gradients in a multi-layer or recurrent network, there is really only one thing we need and that is the *chain rule*.

Backprop in a three-layer network



Suppose we have inputs $\mathbf{x} = [x_1, ..., x_m]^T$, outputs $\mathbf{y} = [y_1, ..., y_n]^T$, hidden units, $\mathbf{h} = [h_1, ..., h_l]^T$ and target outputs, $\mathbf{t} = [t_1, ..., t_n]^T$.

Let's derive backprop on the board...

Backprop in a three-layer network

Following our derivation, we get:

$$\Delta W_{yij} = -\eta \sum_{k=1}^{K} (y_i^k - t_i^k) y_i^k (1 - y_i^k) h_j^k \tag{1}$$

$$\Delta W_{hj\ell} = -\eta \sum_{k=1}^{K} [\sum_{i=1}^{n} (y_i^k - t_i^k) y_i^k (1 - y_i^k) W_{yij}] h_j^k (1 - h_j^k) x_\ell^k \tag{2}$$

Backprop in a three-layer network

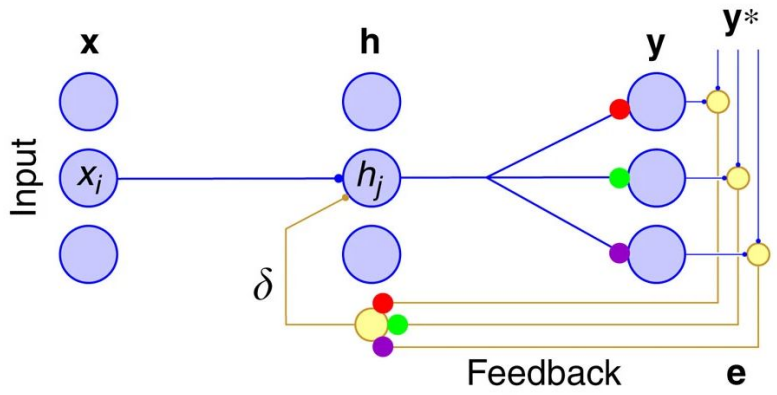
Note: this essentially says that each hidden layer neuron needs to integrate the errors from each output neuron, multiplied by the forward synaptic weight. This is the "error backpropagation" step. It is close to being biologically plausible, but not *quite*.

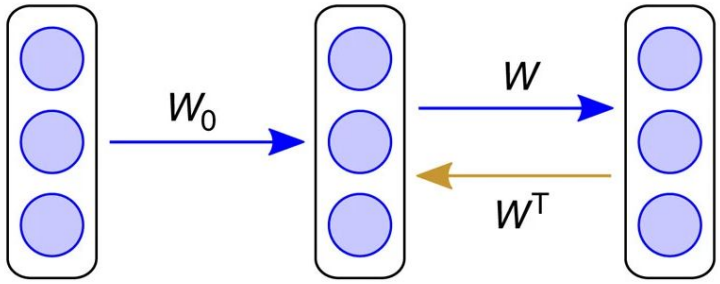
$$\Delta W_{hj\ell} = -\eta \sum_{k=1}^{K} [\sum_{i=1}^{n} (y_i^k - t_i^k) y_i^k (1 - y_i^k) W_{yij}] h_j^k (1 - h_j^k) x_\ell^k \qquad (3)$$

Feedback alignment

What's not biologically plausible about backprop?

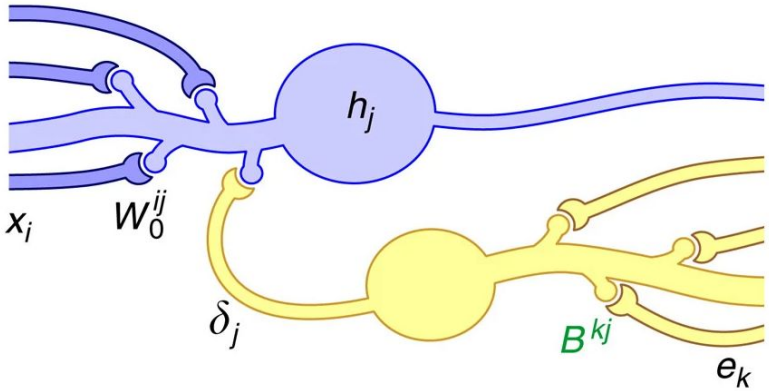
There are a few potential issues, but the really obvious one is the fact that the errors need to be multiplied by the forward synaptic weights. *That implies a feedback pathway for the errors with synapses that are perfectly symmetric to the feedforward pathway!*

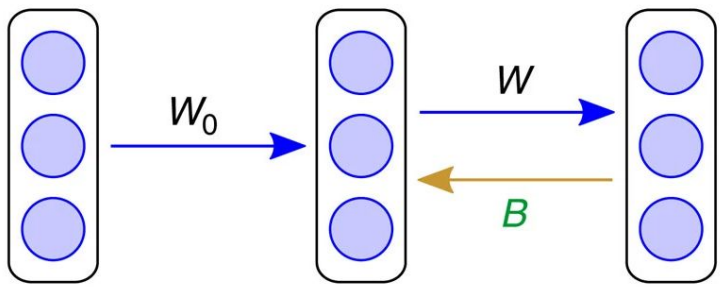




What if we just used random weights?

Lillicrap et al. (2016) demonstrated that, surprisingly, it all works well if you just use random feedback weights!



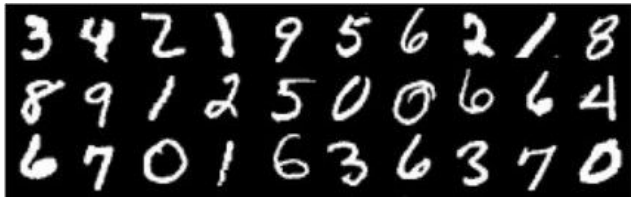


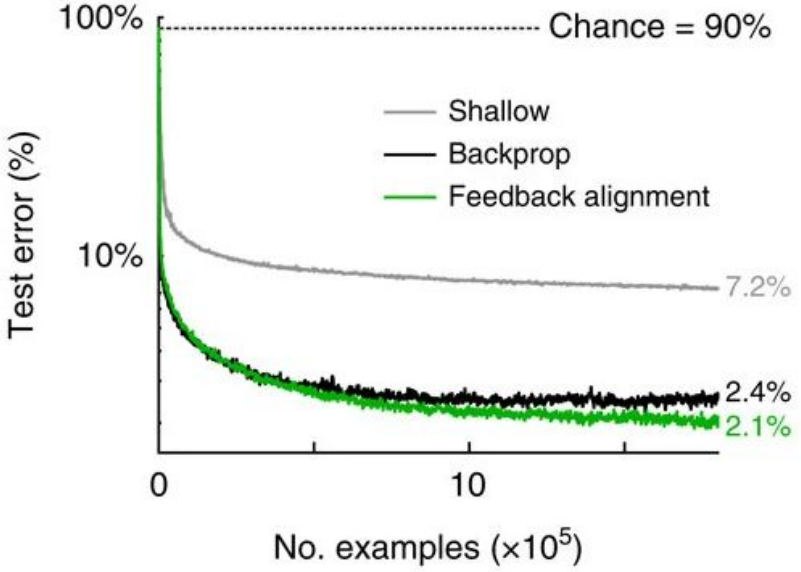
Feedback alignment

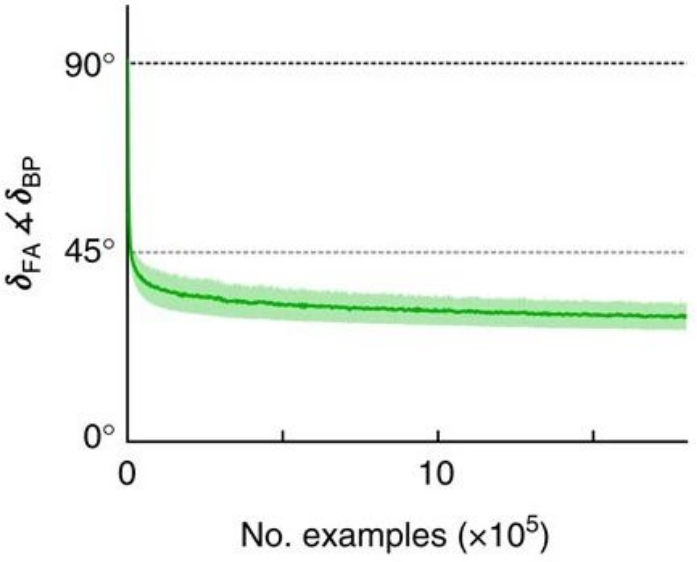
This gives us the **feedback alignment algorithm**, which is just like backprop but uses random, fixed feedback weights:

$$\Delta W_{hj\ell} = -\eta \sum_{k=1}^{K} [\sum_{i=1}^{n} (y_i^k - t_i^k) y_i^k (1 - y_i^k) B_{ji}] h_j^k (1 - h_j^k) x_\ell^k \tag{4}$$

Feedback alignment







Kolen-Pollack

Learning the feedback weights

Practice has shown that random feedback weights are fine for simple problems, but not so great for more complex problems. Can we learn symmetric feedback weights?

This is the solution provided by the **Kolen & Pollack** (1994) algorithm!

The Kolen-Pollack algorithm

Kolen & Pollack (1994) observed that if you update two sets of synaptic weights, W and B, with the same term, plus some history, the weights will eventually converge:

$$\Delta W = A - \lambda W$$
$$\Delta B = A - \lambda B$$

Let's derive this on the board...

The Kolen-Pollack algorithm

For training a neural network, there is an obvious candidate for A, namely, the integrated error terms:

$$\Delta W = A - \lambda W$$

$$\Delta B = A - \lambda B$$

$$A = [\sum_{i=1}^{n} (y_i^k - t_i^k) y_i^k (1 - y_i^k) B_{ji}]$$

Summary

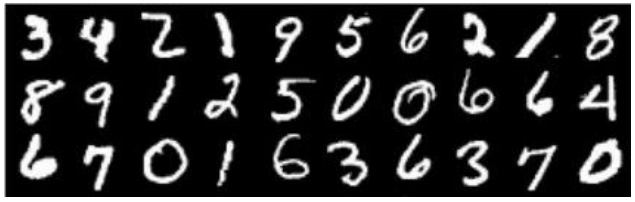
- Learning means getting better. We can quantify this with loss functions.
- Randomly changing your network to see if it gets better is a poor way to learn.
- Gradient descent is a nice, linear approximation to more sophisticated means for finding parameters that minimize a loss function.
- Gradient descent in a multi-layer or recurrent network presents us with the credit assignment problem, though.

Summary

- Backpropagation-of-error is a specific form of gradient descent that can be used to solve the credit assignment problem for multi-layer or recurrent networks.
- Backprop assumes biologically implausible things, though, like a symmetric feedback pathway.
- The credit assignment problem can be solved in a more biologically realistic way using feedback alignment (random feedback weights) or Kolen-Pollack (learned weight symmetry).
- We still do not know how the brain actually solves the credit assignment problem!

Assignment

In your assignment, you will train a multi-layer network to recognize MNIST images using weight perturbation, backprop, feedback alignment, and Kolen-Pollack.



The only piece of code you need to do is the calculation of the weight update - everything else is taken care of in the Python notebook.

Let's get started!