Introduction to neural networks

Herman Kamper

2023-03, CC BY-SA 4.0

Preliminaries

Vector and matrix derivatives

Resources

- A note on vector and matrix calculus
- A video lecture [slides]

Brief summary of vector and matrix derivatives

We will use the denominator layout, where the shape of $\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}}$ matches that of $\mathbf{X}$. (The other option is the Jacobian layout, where the shape of $\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}}$ matches $\mathbf{X}^\top$.)

Derivative of a scalar function $f : \mathbb{R}^N \to \mathbb{R}$ with respect to vector $\mathbf{x} \in \mathbb{R}^N$:

$$
\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \triangleq
\begin{bmatrix}
\frac{\partial f(\mathbf{x})}{\partial x_1} \\
\frac{\partial f(\mathbf{x})}{\partial x_2} \\
\vdots \\
\frac{\partial f(\mathbf{x})}{\partial x_N}
\end{bmatrix}
$$

Derivative of a vector function $\boldsymbol{f} : \mathbb{R}^N \to \mathbb{R}^M$, where $\boldsymbol{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) & f_2(\mathbf{x}) & \cdots & f_M(\mathbf{x}) \end{bmatrix}^\top$, with respect to vector $\mathbf{x} \in \mathbb{R}^N$:

$$
\frac{\partial \boldsymbol{f}(\mathbf{x})}{\partial \mathbf{x}} \triangleq
\begin{bmatrix}
\frac{\partial \boldsymbol{f}(\mathbf{x})}{\partial x_1} \\
\frac{\partial \boldsymbol{f}(\mathbf{x})}{\partial x_2} \\
\vdots \\
\frac{\partial \boldsymbol{f}(\mathbf{x})}{\partial x_N}
\end{bmatrix}
=
\begin{bmatrix}
\frac{\partial f_1(\mathbf{x})}{\partial x_1} & \frac{\partial f_2(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial x_1} \\
\frac{\partial f_1(\mathbf{x})}{\partial x_2} & \frac{\partial f_2(\mathbf{x})}{\partial x_2} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial x_2} \\
\vdots & \vdots & \vdots & \vdots \\
\frac{\partial f_1(\mathbf{x})}{\partial x_N} & \frac{\partial f_2(\mathbf{x})}{\partial x_N} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial x_N}
\end{bmatrix}
$$

Derivative of a scalar function $f : \mathbb{R}^{M \times N} \to \mathbb{R}$ with respect to matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$:

$$\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}} \triangleq \begin{bmatrix} \frac{\partial f(\mathbf{X})}{\partial X_{1,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{1,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{1,N}} \\ \frac{\partial f(\mathbf{X})}{\partial X_{2,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{2,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{2,N}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f(\mathbf{X})}{\partial X_{M,1}} & \frac{\partial f(\mathbf{X})}{\partial X_{M,2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial X_{M,N}} \end{bmatrix}$$

Using the above definitions, we can generalise the chain rule. Given $\mathbf{u} = \boldsymbol{h}(\mathbf{x})$ (i.e. $\mathbf{u}$ is a function of $\mathbf{x}$) and $\boldsymbol{g}$ is a vector function of $\mathbf{u}$, the vector-by-vector chain rule states:

$$\frac{\partial \boldsymbol{g}(\mathbf{u})}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \frac{\partial \boldsymbol{g}(\mathbf{u})}{\partial \mathbf{u}}$$

This generalised chain rule comes from the chain rule for multivariate functions. In the scalar case where g depends on u_1 and u_2, which in turn depends on x, we have (Deisenroth et al. 2020, Sec. 5.2.2):
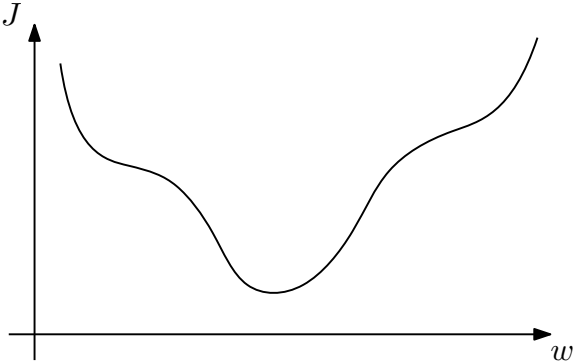
$$\frac{\partial g}{\partial x} = \frac{\partial u_1}{\partial x} \frac{\partial g}{\partial u_1} + \frac{\partial u_2}{\partial x} \frac{\partial g}{\partial u_2}$$

Gradient descent

Resources

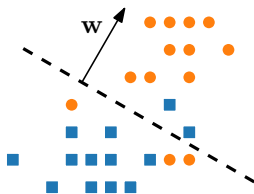- A video lecture [slides]

Brief summary of gradient descent

Logistic regression, softmax, basis functions

Resources

- A video lecture on basis functions in linear regression [slides]
- A playlist of video lectures on binary and softmax regression [slides1, slides2]

Brief summary of binary logistic regression

Model structure:

$$P_{\mathbf{w}}(y = 1|\mathbf{x}) = f_{\mathbf{w}}(\mathbf{x})$$

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

With labels $y \in \{0, 1\}$, minimise the negative log likelihood (NLL):

$$J(\mathbf{w}) = -\log \prod_{n=1}^{N} P_{\mathbf{w}}(y^{(n)}|\mathbf{x}^{(n)})$$

$$= -\sum_{n=1}^{N} \left[y^{(n)} \log f_{\mathbf{w}}(\mathbf{x}^{(n)}) + (1 - y^{(n)}) \log \left(1 - f_{\mathbf{w}}(\mathbf{x}^{(n)})\right) \right]$$

Gradient:

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = -\sum_{n=1}^{N} \left(y^{(n)} - f_{\mathbf{w}}(\mathbf{x}^{(n)}) \right) \mathbf{x}^{(n)}$$
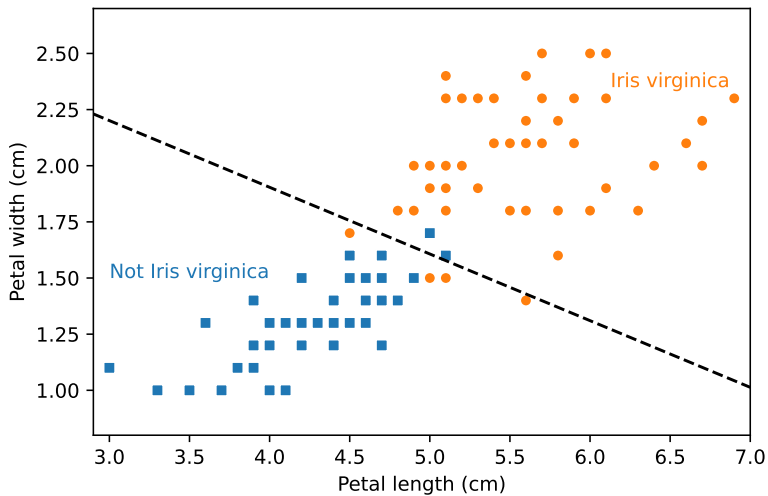
How would you modify the above to do binary logistic regression with basis functions?

Binary classification of irises
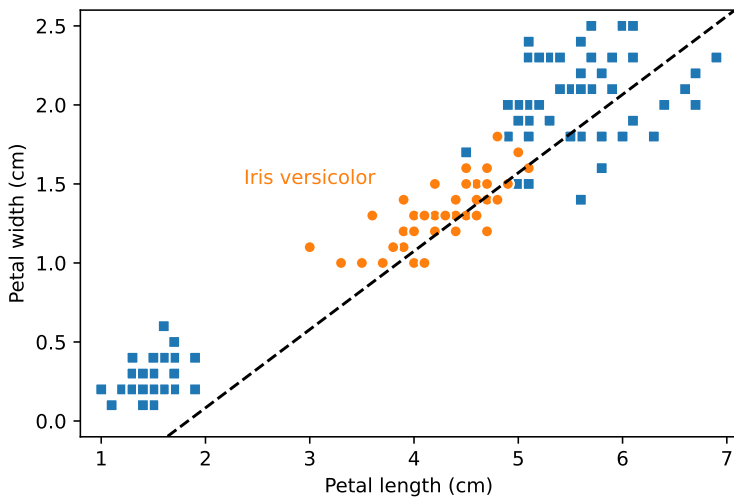
We want to classify irises based on their leaves:[1]

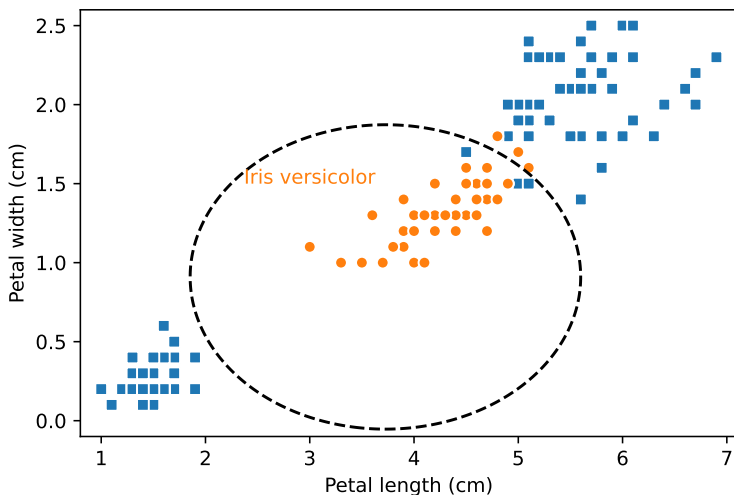Binary logistic regression for classifying virginicas:

[1]Figure from Wikipedia.

Binary logistic regression for classifying versicolors is more difficult, since you can't separate them linearly:

How would you address this? Use basis functions:

Binary logistic regression with basis functions as a neural network

You have already seen a neural network, you just didn't fit it:

- Linear logistic regression:

$$f_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}}$$

- Nonlinear logistic regression:

$$f_{\mathbf{w}}(\boldsymbol{\phi}(\mathbf{x})) = \sigma(\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})) = \frac{1}{1 + e^{-\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x})}}$$

We can represent binary logistic regression with basis functions as a neural network:
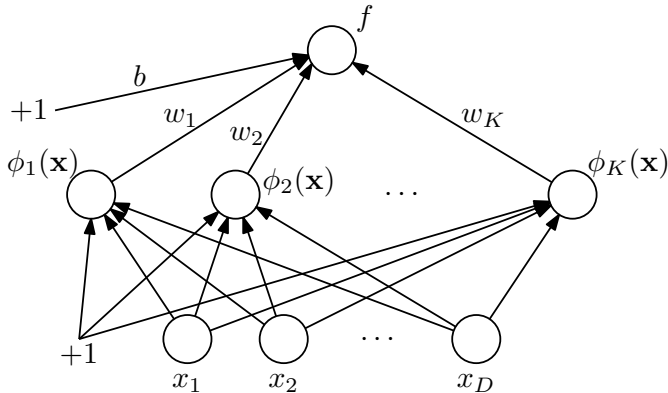
Instead of designing the basis functions $\boldsymbol{\phi}(\mathbf{x})$ by hand, can we learn these features instead?

We can set each basis function value as:

$$\phi_k(\mathbf{x}) = \sigma(\mathbf{w}_k^{[1]\top}\mathbf{x} + b_k^{[1]})$$

or more generally as a nonlinear function of the input:

$$\phi_k(\mathbf{x}) = g(\mathbf{w}_k^{[1]\top}\mathbf{x} + b_k^{[1]})$$

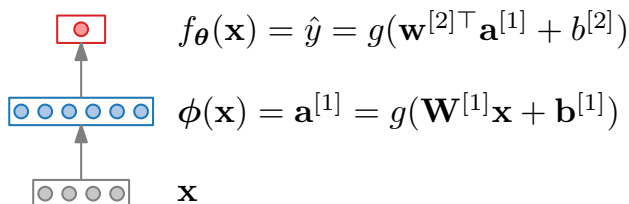where $g(\cdot)$ is some nonlinear function and we learn the parameters $\mathbf{w}^{[1]}$ and $b^{[1]}$ for all features $k = 1$ to K.

Common options for the nonlinearity include:

- Sigmoid

- tanh

- Rectified linear unit (ReLU)

We could express this neural network more compactly in vector form:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \hat{y} = g(\mathbf{w}^{[2]\top}\mathbf{a}^{[1]} + b^{[2]})$$

$$\phi(\mathbf{x}) = \mathbf{a}^{[1]} = g(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]})$$

$$\mathbf{x}$$

This network has a single hidden layer with a binary classification output layer. It is a special case of a general class of neural networks: *feedforward neural networks*, also called *multilayer perceptrons*.

The crucial difference between logistic regression with basis functions and what we will do here is that we fit *all* the parameters from data:

$$\boldsymbol{\theta} = \left\{ \mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{w}^{[2]}, b^{[2]} \right\}$$

Our example: We will use the specific binary feedforward network as our running example in this note, but the principles are general.

Why is it called a neural network?

Very very very loosely inspired by structures in the brain:[2]
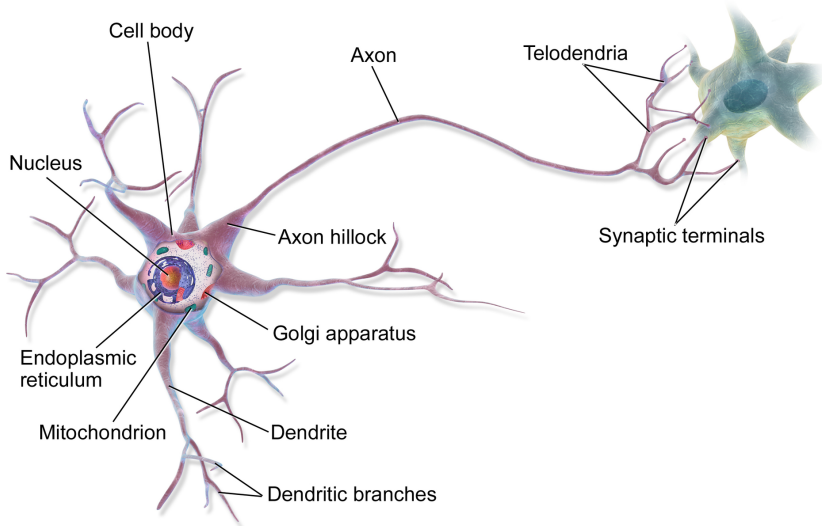
Some people hate it when this connection is mentioned, but:

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an

---

[2]Figure from Wikipedia.

Backpropagation without forks

How do we fit the parameters in our example?

As usual: Use gradient descent to minimise the NLL.

If we have a single training item $(\mathbf{x}^{(n)}, y^{(n)})$:

$$J(\boldsymbol{\theta}) = - \left[y^{(n)} \log \hat{y}^{(n)} + (1 - y^{(n)}) \log(1 - \hat{y}^{(n)}) \right]$$

For gradient descent: Need $\frac{\partial J}{\partial \mathbf{u}}$, where $\mathbf{u}$ is each of the parameters:

$$\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{w}^{[2]}, b^{[2]}$$

You can try and get these in some arbitrary way, but the backpropagation algorithm gives a principled procedure to obtain these gradients.

It essentially applies the chain rule in an order where previously computed results are reused.

The backpropagation algorithm (without forks)

- Represent your neural network as a computational graph. We use the convention where each node represents an operation and each edge a variable.

- **Forward pass:** Start at the inputs and calculate the output of each operation (node) in the graph. Store these values for use in the backward pass.

- **Backward pass:** Start at the output of the graph and move backwards. For each operation, do the following:

  (a) Determine and calculate the derivative of the output variable w.r.t. each of the input variables to the operation.
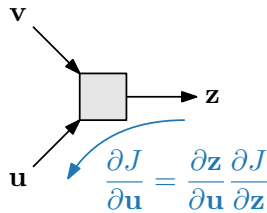
$$\frac{\partial J}{\partial \mathbf{u}} = \frac{\partial \mathbf{z}}{\partial \mathbf{u}} \frac{\partial J}{\partial \mathbf{z}}$$

For this operation, we would determine $\dfrac{\partial \mathbf{z}}{\partial \mathbf{u}}$ and $\dfrac{\partial \mathbf{z}}{\partial \mathbf{v}}$.

  (b) For each input variable $\mathbf{u}$, set[3]

$$\boldsymbol{\delta}_{\mathbf{u}} = \frac{\partial J}{\partial \mathbf{u}} = \frac{\partial \mathbf{z}}{\partial \mathbf{u}} \frac{\partial J}{\partial \mathbf{z}}$$

where $\mathbf{z}$ is the output of the operation taking $\mathbf{u}$ as input.

- Use the calculated derivatives to do take a gradient step to update the parameters. Repeat from the forward pass.

---

[3]We will see later that, if we have forks, we actually accumulate terms like this for every operation taking $\mathbf{u}$ as input. But in our example we don't have forks.

Backpropagation for our example

Our example:

- Feedforward neural network with one hidden layer and a binary output layer.

- Single training item: $(\mathbf{x}^{(n)}, y^{(n)})$

Computational graph

J

(a) $\quad -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$

$\hat{y}$

(b) $\quad g(\cdot)$

$z^{[2]} = \mathbf{w}^{[2]\top} \mathbf{a}^{[1]} + b^{[2]}$

(c) $\quad \mathbf{w}^{[2]} \quad b^{[2]}$

$\mathbf{a}^{[1]}$

(d) $\quad g(\cdot)$

$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \mathbf{x} + \mathbf{b}^{[1]}$

(e) $\quad \mathbf{W}^{[1]} \quad \mathbf{b}^{[1]}$

$\mathbf{x}$
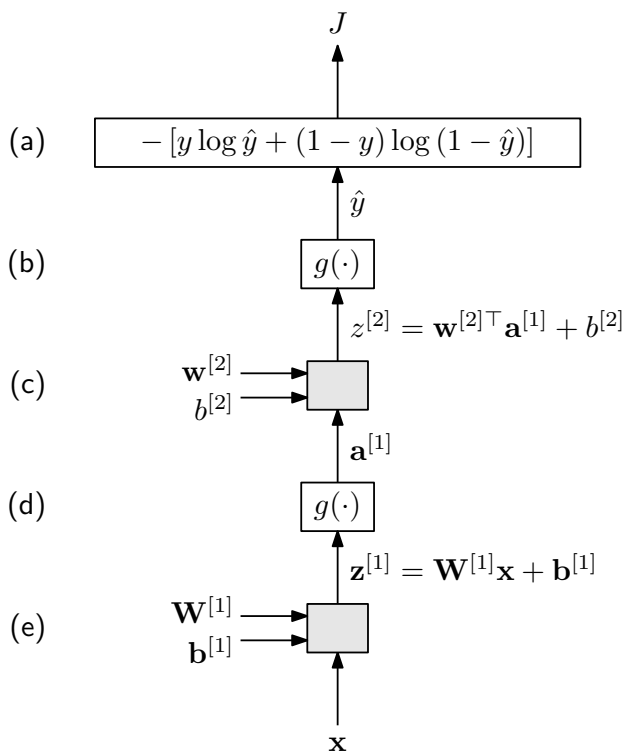
Forward pass

Easy.

Backward pass

(a)
$$\frac{\partial J}{\partial \hat{y}} = -\frac{\partial}{\partial \hat{y}} \left[y \log \hat{y} + (1 - y) \log (1 - \hat{y}) \right]$$
$$= -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}$$
$$\delta_{\hat{y}} = \left. \frac{\partial J}{\partial \hat{y}} \right|_{\substack{\mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}} = -\frac{y^{(n)}}{\hat{y}^{(n)}} + \frac{1 - y^{(n)}}{1 - \hat{y}^{(n)}}$$

(b)
$$\frac{\partial \hat{y}}{\partial z^{[2]}} = g'(z^{[2]})$$
$$\delta_{z^{[2]}} = \left. \frac{\partial J}{\partial z^{[2]}} \right|_{\substack{\mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}} = \left[\frac{\partial \hat{y}}{\partial z^{[2]}} \frac{\partial J}{\partial \hat{y}} \right]_{\substack{\mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}}$$
$$= \left. \frac{\partial \hat{y}}{\partial z^{[2]}} \right|_{\substack{\mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}} \delta_{\hat{y}}$$

(c) Using vector calculus:[4]
$$\frac{\partial z^{[2]}}{\partial \mathbf{w}^{[2]}} = \frac{\partial}{\partial \mathbf{w}^{[2]}} \left(\mathbf{w}^{[2]\top} \mathbf{a}^{[1]} + b^{[2]} \right)$$
$$= \mathbf{a}^{[1]}$$
$$\boldsymbol{\delta}_{\mathbf{w}^{[2]}} = \frac{\partial J}{\partial \mathbf{w}^{[2]}} = \frac{\partial z^{[2]}}{\partial \mathbf{w}^{[2]}} \frac{\partial J}{\partial z^{[2]}}$$
$$= \frac{\partial z^{[2]}}{\partial \mathbf{w}^{[2]}} \delta_{z^{[2]}}$$
$$\boldsymbol{\delta}_{b^{[2]}} = \frac{\partial z^{[2]}}{\partial b^{[2]}} \delta_{z^{[2]}}$$
$$\boldsymbol{\delta}_{\mathbf{a}^{[1]}} = \frac{\partial z^{[2]}}{\partial \mathbf{a}^{[1]}} \delta_{z^{[2]}}$$

---

[4]I am dropping the bar now because I am lazy.

(d) Similar to (b).

(e) Be careful with this one since it involves a matrix (check the shapes!):

$$\boldsymbol{\delta}_{\mathbf{W}^{[1]}} = \frac{\partial J}{\partial \mathbf{W}^{[1]}}$$
$$= \delta_{\mathbf{z}^{[1]}} \mathbf{x}^{\top}$$

See my YAIT backprop note for details on the derivation for this last operation.

Numerical values for the gradients at our current estimate

Why did I write the bar in the following?

$$\delta_{\hat{y}} = \frac{\partial J}{\partial \hat{y}}\bigg|_{\substack{\mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}}$$

Actually I should have written

$$\delta_{\hat{y}} = \frac{\partial J}{\partial \hat{y}}\bigg|_{\substack{\boldsymbol{\theta}=\hat{\boldsymbol{\theta}}^{(m)} \\ \mathbf{x}=\mathbf{x}^{(n)} \\ y=y^{(n)}}}$$

Why? Because we are calculating the numerical value of the gradients at our particular current estimate of the parameters $\hat{\boldsymbol{\theta}}^{(m)}$ in iteration m. In the forward pass of the next iteration $m + 1$, the numerical values for the gradients at this new point will be different. Also, if we change the data (e.g. for a different mini-batch), then the numerical values of the gradients will also change. In short: We are at some specific point of J, and we are taking a optimal gradient step from that specific point. Relate this to the brief summary of gradient descent given earlier.

Classification of irises using our neural network
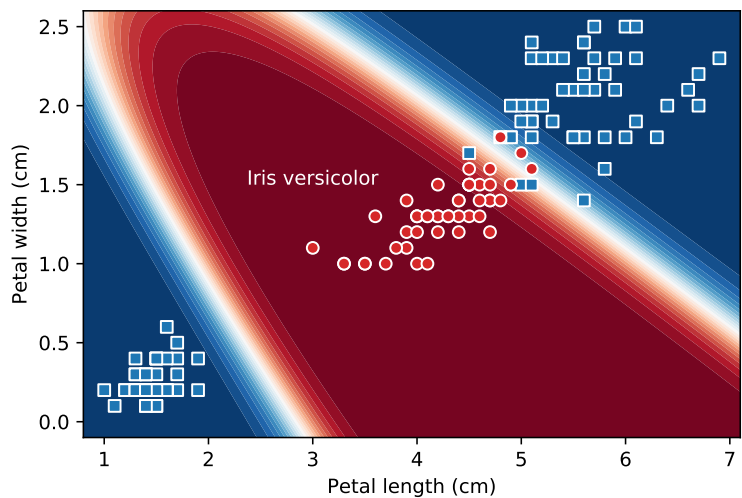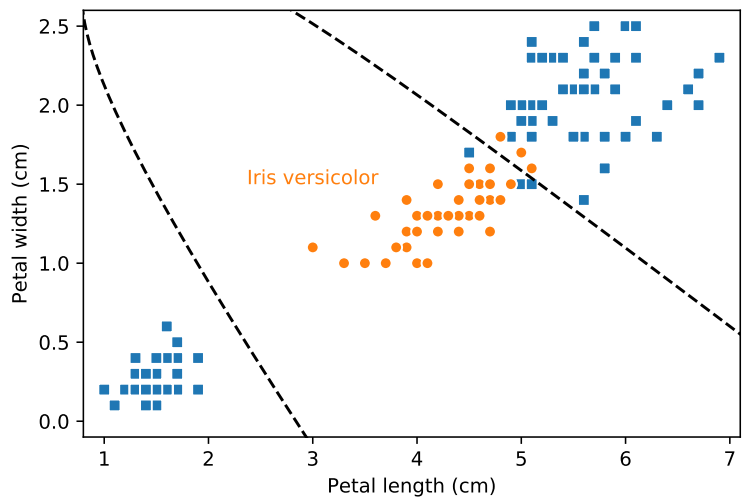
Multilayer feedforward neural network

Above we looked at the specific example of a binary feedforward neural network with one hidden layer. Let's go a bit more general.
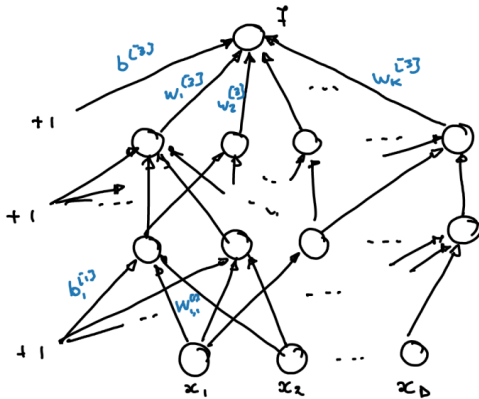
$$\hat{y} = g\left(\underline{w}^{(3)T} \underline{a} + b^{(3)}\right)$$

Hidden layer 2:
$$\underline{a} = g\left(\underline{w}^{[2]} \underline{a}^{[1]} + b^{[2]}\right)$$

Hidden layer 1:
$$\underline{a}^{[1]} = g\left(\underline{w}^{[1]} \underline{x} + \underline{b}^{[1]}\right)$$

$$\frac{\partial J}{\partial w_{i,i'}^{[j]}} = ?$$

Vector diagram:
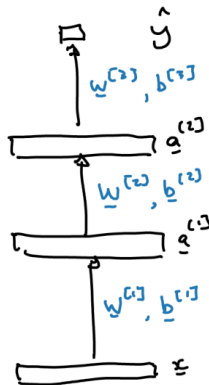
A feedforward neural network with L hidden layers using a squared loss function:
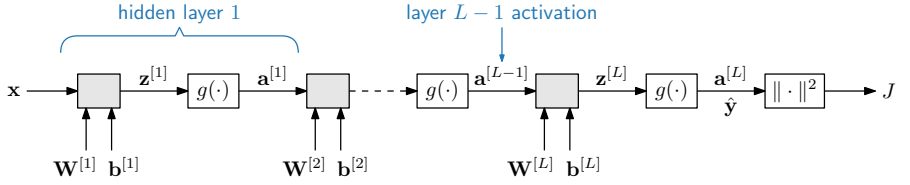
For arbitrary layer l:
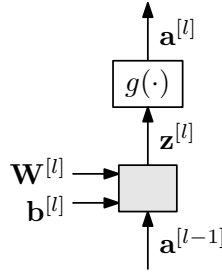
$$\boldsymbol{\delta}_{\mathbf{z}^{[l]}} = \boldsymbol{\delta}_{\mathbf{a}^{[l]}} \odot g'(\mathbf{z}^{[l]})$$

$$\boldsymbol{\delta}_{\mathbf{b}^{[l]}} = \boldsymbol{\delta}_{\mathbf{z}^{[l]}}$$

$$\boldsymbol{\delta}_{\mathbf{W}^{[l]}} = \boldsymbol{\delta}_{\mathbf{z}^{[l]}} \mathbf{a}^{[l-1]\top}$$

$$\boldsymbol{\delta}_{\mathbf{a}^{[l-1]}} = \mathbf{W}^{[l]\top} \boldsymbol{\delta}_{\mathbf{z}^{[l]}}$$

In more traditional explanations, these equations are combined:

$$\boldsymbol{\delta}_{\mathbf{z}^{[l]}} = \left(\mathbf{W}^{[l+1]\top} \boldsymbol{\delta}_{\mathbf{z}^{[l+1]}} \right) \odot g'(\mathbf{z}^{[l]})$$

This highlights the recursive nature of backpropagation.

Why this (cool) graph formulation?

More traditional explanations of neural networks don't use this graph formulation and instead derives things more specifically for particular architectures. This simplifies things (a bit) and is very concrete. But this type of conventional explanation is also quite rigid and then hides the flexibility that comes with the graph formulation.

Adding additional structure is easy:

- As long as we know the derivative of a single operation (node), the gradient computation is fully specified by the graph.

- Each node just needs to know how to compute its output and how to compute the gradient w.r.t. its inputs given the gradient w.r.t. its output.

```python
class MultiplyGate():

    def forward(x, y):
        z = x*y
        self.x = x
        self.y = y
        return z

    def backward(delta_z):
        delta_x = self.y * delta_z
                # dz/dx  * dJ/dz
        delta_y = self.x * delta_z
                # dz/dy  * dJ/dz
        return [delta_x, delta_y]
```

Automatic differentiation

We've looked at this specifically for neural networks, but this is actually an example of the more general methodology of automatic differentiation.

Modern deep learning frameworks like PyTorch and Tensorflow (or the older Theano) make use of this graph formulation with automatic differentiation. And, fortunately, some good programmer has probably implemented most of the nodes/blocks/operations for you already (forward and gradient).

Why then study this if the software can do it?
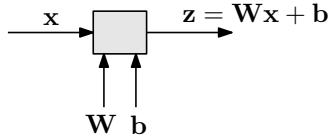
So why did we study how to do this, if software can give us all the derivatives automatically?

- In some very simple cases, you might not want to have to rely on (the bulky) PyTorch or Tensorflow. E.g. the gradients for word2vec is relatively straightforward.

- Sometimes you might want to introduce a new computational operation and then you might need to implement the gradient computation.

- More often: You are hacking parts of the gradient computation for an existing block and need to modify it.

Backpropogation (now general)

A summary of derivatives for common blocks

Some of these are derived above, others are derived in YAIT backprop:

$$\delta_{\mathbf{b}} = \frac{\partial \mathbf{z}}{\partial \mathbf{b}} \, \delta_{\mathbf{z}}$$
$$= \mathbf{I} \, \delta_{\mathbf{z}} = \delta_{\mathbf{z}}$$

$$\delta_{\mathbf{W}} = \delta_{\mathbf{z}} \, \mathbf{x}^{\top}$$

$$\delta_{\mathbf{x}} = \frac{\partial \mathbf{z}}{\partial \mathbf{x}} \, \delta_{\mathbf{z}}$$
$$= \mathbf{W}^{\top} \, \delta_{\mathbf{z}}$$

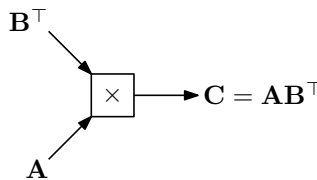$$\delta_{\mathbf{B}} = \delta_{\mathbf{C}}^{\top} \mathbf{A}$$

$$\delta_{\mathbf{A}} = \delta_{\mathbf{C}} \mathbf{B}$$

A general notation

In all of the above cases, for arbitrary[5] variable U going in to operation with output Z, the error signal δ_{U} is obtained as some kind of product between $\frac{\partial \mathsf{Z}}{\partial \mathsf{U}}$ and $\frac{\partial J}{\partial \mathsf{Z}} = \delta_{\mathsf{Z}}$. For instance:

- With vectors as inputs and outputs:[6] $\delta_{\mathbf{b}} = \dfrac{\partial \mathbf{z}}{\partial \mathbf{b}} \, \delta_{\mathbf{z}}$

- But sometimes the order between $\dfrac{\partial \mathsf{Z}}{\partial \mathsf{U}}$ and δ_{Z} flips: $\delta_{\mathbf{W}} = \delta_{\mathbf{z}} \, \mathbf{x}^{\top}$

- Or we have to also take the transpose: $\delta_{\mathbf{B}} = \delta_{\mathbf{C}}^{\top} \mathbf{A}$

---

[5]Scalar, vector, matrix or tensor.
[6]This is just generalised chain rule.

Based on this observation, Zhang et al. (2021) gives a very nice way to capture all of these details with a new operator, prod:

$$\boldsymbol{\delta}_{\mathsf{U}} = \frac{\partial J}{\partial \mathsf{U}} = \mathrm{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{U}}, \frac{\partial J}{\partial \mathsf{Z}}\right)$$
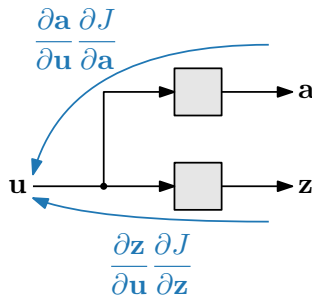$$= \mathrm{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{U}}, \boldsymbol{\delta}_{\mathsf{Z}}\right)$$

This looks very similar to the the generalised chain rule for vectors, but can now be applied to scalars, vectors, matrices or tensors without a bloaty notation. The prod operator captures all the necessary details: transpositions, swapping input positions and anything else that we need to deal with.

About forks

In the title of the previous backpropagation section I explicitly noted that we were dealing with networks without forks, i.e. every variable serves as the input to only a single operation (node in the computational graph). What happens when we have forks?

From the generalised chain rule: We need to accumulate all the gradients for a variable.

To be concrete, for the graph fragment:

we will have the accumulator

$$\boldsymbol{\delta}_{\mathbf{u}} = \frac{\partial \mathbf{z}}{\partial \mathbf{u}} \frac{\partial J}{\partial \mathbf{z}} + \frac{\partial \mathbf{a}}{\partial \mathbf{u}} \frac{\partial J}{\partial \mathbf{a}}$$
$$= \frac{\partial \mathbf{z}}{\partial \mathbf{u}} \boldsymbol{\delta}_{\mathbf{z}} + \frac{\partial \mathbf{a}}{\partial \mathbf{u}} \boldsymbol{\delta}_{\mathbf{a}}$$

Using the general notation and for the case where this fragment has variables that might be matrices or tensors, we would have

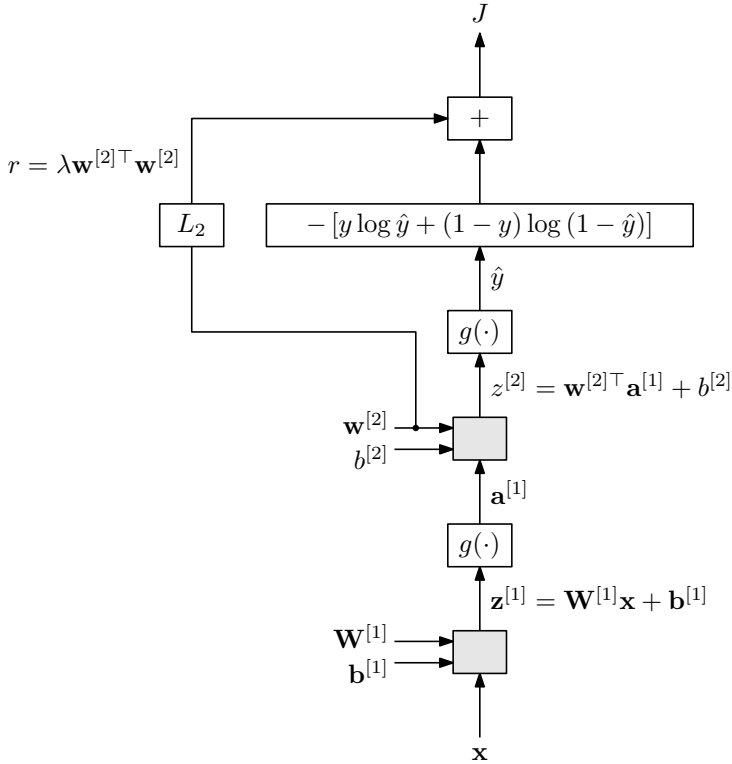$$\boldsymbol{\delta}_{\mathsf{U}} = \mathrm{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{U}}, \boldsymbol{\delta}_{\mathsf{Z}}\right) + \mathrm{prod}\left(\frac{\partial \mathsf{A}}{\partial \mathsf{U}}, \boldsymbol{\delta}_{\mathsf{A}}\right)$$

Example: L_2 regularisation

A real example of where this might happen is if we regularise our model weights. If we modify our binary feedforward neural network to include L_2 regularisation on just the $\mathbf{w}^{[2]}$ weight vector:

$$J(\boldsymbol{\theta}) = -\left[y^{(n)} \log f_{\boldsymbol{\theta}}(x^{(n)}) + (1 - y^{(n)}) \log \left(1 - f_{\boldsymbol{\theta}}(x^{(n)})\right)\right] + \lambda \mathbf{w}^{[2]\top} \mathbf{w}^{[2]}$$

then our computational graph would look like this:

Then the accumulator would change to

$$\begin{aligned}
\boldsymbol{\delta}_{\mathbf{w}^{[2]}} = \frac{\partial J}{\partial \mathbf{w}^{[2]}} &= \frac{\partial z^{[2]}}{\partial \mathbf{w}^{[2]}} \frac{\partial J}{\partial z^{[2]}} + \frac{\partial r}{\partial \mathbf{w}^{[2]}} \frac{\partial J}{\partial r} \\
&= \frac{\partial z^{[2]}}{\partial \mathbf{w}^{[2]}} \delta_{z^{[2]}} + \frac{\partial r}{\partial \mathbf{w}^{[2]}} \delta_r \\
&= \mathbf{a}^{[1]} \delta_{z^{[2]}} + 2\lambda \mathbf{w}^{[2]} \delta_r
\end{aligned}$$

The backpropagation algorithm (now with forks)

- Initialisation: Set accumulators to zero for all input variables:

$$\delta_U \leftarrow 0 \text{ for every } U$$

- **Forward pass:** Start at the inputs and calculate the output of each operation (node) in the graph.

- **Backward pass:** Start at the output of the graph and move backwards. For each operation, do the following:

    (a) Determine and calculate the derivative of the output variable w.r.t. each of the input variables to the operation.
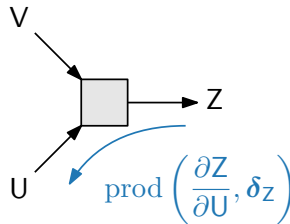
    For this operation, we would determine $\dfrac{\partial Z}{\partial U}$ and $\dfrac{\partial Z}{\partial V}$.

    (b) For each input variable U, add

    $$\mathrm{prod}\left(\frac{\partial Z}{\partial U}, \frac{\partial J}{\partial Z}\right)$$

    to its accumulator, i.e.

    $$\delta_U \leftarrow \delta_U + \mathrm{prod}\left(\frac{\partial Z}{\partial U}, \delta_Z\right)$$

    Note that in the backward pass you can only consider the operation with output Z once its associated accumulator δ_Z is finalised, i.e. all operations taking Z as input has been backpropped. This might affect the backprop order. Stated differently, when updating δ_U as above, δ_Z needs to be final.

On the NLL and cross entropy loss for multiclass classification

Why do we often say we use the **cross entropy loss** for multiclass classification in frameworks like PyTorch? Aren't we using the NLL? These are actually the same.

Let's consider a neural network where the final linear output $\mathbf{z}^{[L]}$ feeds into a softmax layer (I'm dropping the layer superscript from here onwards):

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{1}{\sum_{j=1}^{K} \exp(z_j)} \begin{bmatrix} \exp(z_1) \\ \exp(z_2) \\ \vdots \\ \exp(z_K) \end{bmatrix} = \text{softmax}(\mathbf{z})$$

The values $\mathbf{z}$ are sometimes called the logits. They can be seen as unnormalised log probabilities.

If we write the target output as a one-hot vector:

$$\mathbf{y}^{(n)} = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^{\top}$$

then we can write the NLL as:

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} \log \frac{\exp(z_k)}{\sum_{j=1}^{K} \exp(z_j)} \\ &= -\sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} \log f_{\boldsymbol{\theta},k}(\mathbf{x}^{(n)}) \\ &= -\sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} \log \hat{y}_k^{(n)} \end{aligned}$$

Now recall the definition of cross entropy from the earlier note:

$$H(\mathbf{p}, \mathbf{q}) \triangleq - \sum_{k=1}^{K} p_k \log_2 q_k$$

So thinking of $\mathbf{y}^{(n)}$ as a discrete probability distribution with all of its mass on a single outcome k, then the NLL can be seen as the cross entropy between the true distribution $\mathbf{y}^{(n)}$ and the predicted distribution $\hat{\mathbf{y}}^{(n)}$.

Unsupervised neural networks: Autoencoders

Principal components analysis:

$$\hat{x} = \underline{W}\, z$$

$$z = \underline{W}^{\mathsf{T}} \underline{x}$$

Autoencoder:

$$a^{(2)}$$

$$a^{(\cdot)} = g(\underline{W}^{(\cdot)} \underline{x} + b)$$

$$J = \sum_{n=1}^{N} \left\| \underline{x}^{(n)} - \hat{\underline{x}}^{(n)} \right\|^2$$
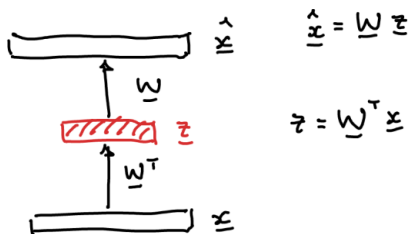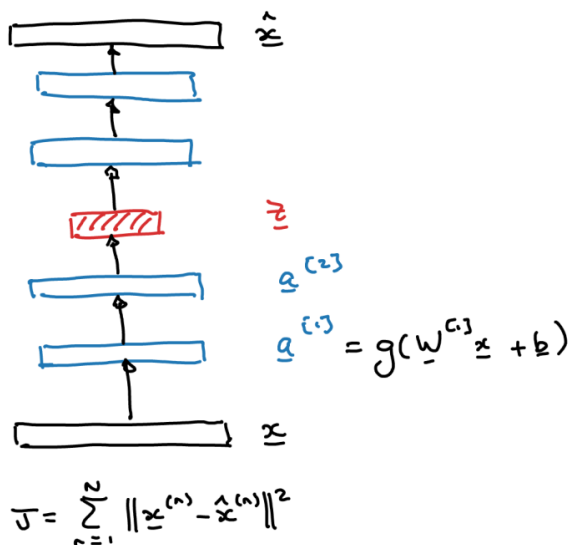
Neural networks in practice

Different names for feedforward neural networks

- Multilayer neural network

- Feedforward neural network (FFNN)

- Multilayer perceptron (MLP)

- Artificial neural network (ANN)

- Deep neural network (DNN)

Developing neural networks is in an art

- Sometimes useful to scale inputs.

- Instead of vanilla gradient descent, we often use advanced forms of mini-batch gradient descent (Adam is popular at the moment).

- Different initialisation strategies, e.g. https://arxiv.org/abs/1811.00293.

- Overfitting: Can combat using standard regularisation, but often rather just use dropout or rely on SGD with early stopping.

- Need to choose number of hidden layers and number of units per layer, and often many more hyperparameters.

- Often make architecture choices (e.g. skip connections) to deal with optimisation problems (e.g. exploding or vanishing gradients—more on this later).

NLP example: Named entity recognition

Named entity recognition (NER): Given an input sentence, find and classify the names according to their named entity types.

Examples:

```
last night Paris Hilton wowed in a sequin gown
            PER    PER

Samuel Quinn was arrested in the Hilton Hotel
PER    PER                        LOC    LOC
                        in Paris in April 1989
                           LOC      DATE DATE
```

Example named entity types:

Tag	Description	Example
PER	People, characters	**Shannon** is a giant of information theory.
ORG	Organisation	The **ICC** is the governing body of cricket.
LOC	Location	**Mt. Sanitas** is in **Sunshine Canyon**.
GPE	Geo-political (countries, states)	Petrol prices are going up in **South Africa**.
DATE	Days, months, years	Micah was born in **April**.

Example applications:

- Tracking mentions of specific entities of interest in documents.

- Question answering: Answers are often named entities.

- Semantic analysis: Sentiment in discussions of some entity.

NER with a neural network

Example sentence:

```
anywhere in Paris museums are great
```

We want to classify the entity of `Paris`.

We use a window of words around the centre word that we want to classify (window length of one here):

$$\mathbf{x}_{\text{win}} = \begin{bmatrix} -\mathbf{x}_{\text{in}}^\top- & -\mathbf{x}_{\text{Paris}}^\top- & -\mathbf{x}_{\text{museums}}^\top- \end{bmatrix}^\top$$
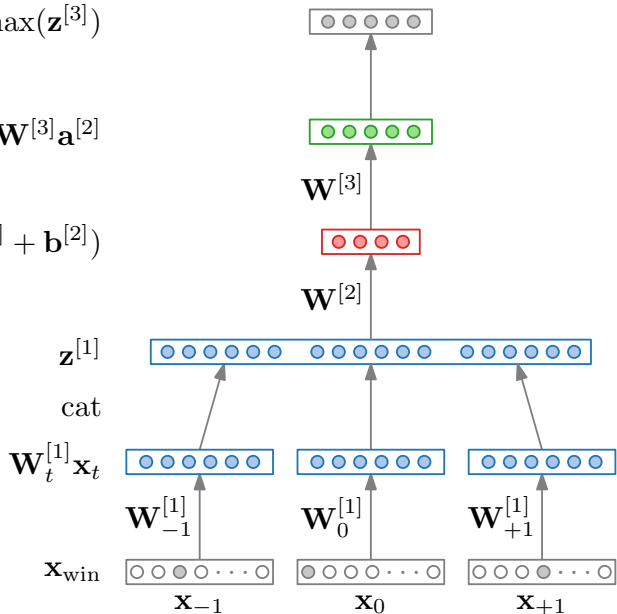
Each element of the window $\mathbf{x}_t$ could be a word embedding, but let's instead have each be a one-hot vector representing the word type (we'll see in a bit what we get when we do this).

We can then use the following neural network for NER:

$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{\text{win}}) = \text{softmax}(\mathbf{z}^{[3]})$$

$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]}\mathbf{a}^{[2]}$$

$$\mathbf{a}^{[2]} = g(\mathbf{W}^{[2]}\mathbf{z}^{[1]} + \mathbf{b}^{[2]})$$

$\mathbf{z}^{[1]}$

cat

$\mathbf{W}_t^{[1]}\mathbf{x}_t$

$\mathbf{W}_{-1}^{[1]}$ $\mathbf{W}_0^{[1]}$ $\mathbf{W}_{+1}^{[1]}$

$\mathbf{x}_{\text{win}}$

$\mathbf{x}_{-1}$ $\mathbf{x}_0$ $\mathbf{x}_{+1}$
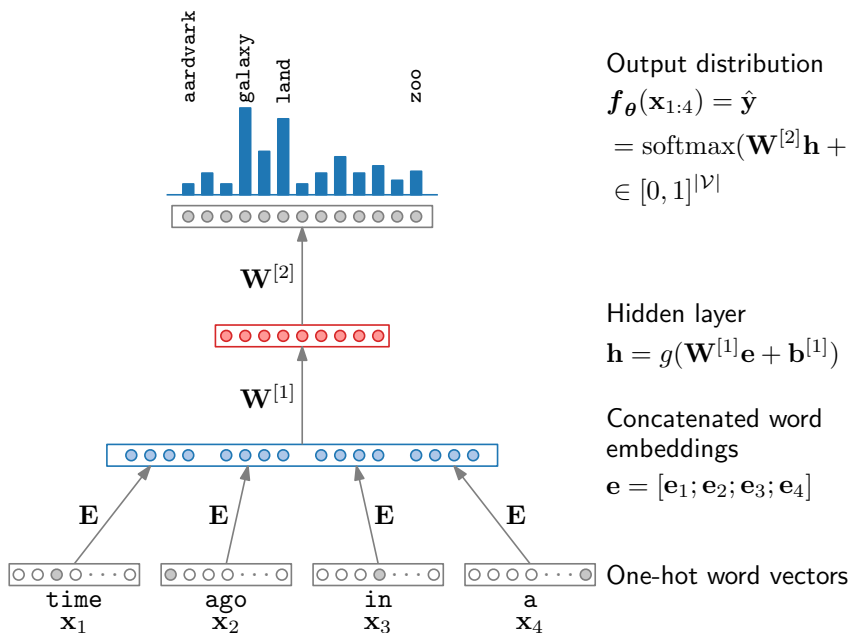
Learning word embeddings and a classifier jointly

- When $\mathbf{x}_t$ is one-hot, what does the operation $\mathbf{W}_t \times \mathbf{x}_t$ give us in terms of $\mathbf{W}_t$? What is $\mathbf{W}_t$ representing?

- We are just looking up embeddings!

- The above neural network is jointly learning to classify ($\mathbf{W}^{[3]}$, $\mathbf{W}^{[2]}$, $\mathbf{b}^{[2]}$) and also three types of word embeddings ($\mathbf{W}^{[1]}_{-1}$, $\mathbf{W}^{[1]}_0$, $\mathbf{W}^{[1]}_{+1}$), all at the same time!

NLP example: Neural language models

This windowed classification approach, where word embeddings are learned jointly with a classifier, also formed the basis of early neural language models (Bengio et al. 2003).

A long long time ago in a ...

Output distribution
$$\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_{1:4}) = \hat{\mathbf{y}}$$
$$= \text{softmax}(\mathbf{W}^{[2]}\mathbf{h} + \mathbf{b}^{[2]})$$
$$\in [0,1]^{|\mathcal{V}|}$$

Hidden layer
$$\mathbf{h} = g(\mathbf{W}^{[1]}\mathbf{e} + \mathbf{b}^{[1]})$$

Concatenated word embeddings
$$\mathbf{e} = [\mathbf{e}_1; \mathbf{e}_2; \mathbf{e}_3; \mathbf{e}_4]$$

One-hot word vectors

Further reading

I would encourage you to go through the much more detailed YAIT backprop notes, which includes more exact details of the gradient calculations given here.

References

Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *JLMR*, 2003.

CS231n: Optimization 2, *Stanford University*, 2023.

M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for Machine Learning*, 1st ed., 2020.

H. Kamper, "Yet another introduction to backpropagation," *Stellenbosch University*, 2022.

C. Manning, "CS224N: Neural net learning, gradients by hand (matrix calculus) and algorithmically (the backpropagation algorithm)," *Stanford University*, 2022.

I. Murray, "MLPR: Backpropagation of derivatives," *University of Edinburgh*, 2018.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, 2021.

A. Ng, "Multi-layer neural network," *Stanford University*, 2013.