Data Science: Principles and Practice

Lecture 7

Guy Emerson

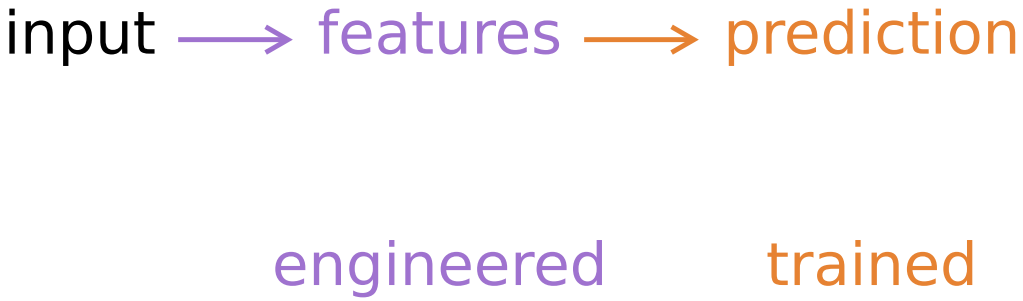
Today's Lecture

- Neural networks:
  - Architectures
  - Training
- Overfitting

Disclaimer: any similarity with biological neural networks is coincidental.

Many data scientists now jump straight for neural network models. Hopefully, the past 6 lectures will help to situate neural nets within a wider range of tools.

Features



Many machine learning models can be broken into two steps: feature extraction, and training. Recall from the practicals how we manually defined features (such as for the housing dataset), and then trained a classifier.

Features

input ⟶ features ⟶ prediction

trained trained

- Engineering at a more abstract level

Neural network models train the features as well. People talk of “end-to-end” training, because all steps are trained, from the input to the output.

Engineering decisions are pushed to a higher level: not in terms of individual features, but in terms of the model architecture.

Feedforward Networks

$$x \mapsto f_1(x) \mapsto f_2(f_1(x))$$

- Linear: $f(x) = Ax$
- but can simplify matrix multiplication $AB = C$
- Nonlinear: $f(x) = g(Ax)$ (g applied componentwise)
- Can approximate any function

A feedforward net applies a sequence of functions to map from the input to the output.

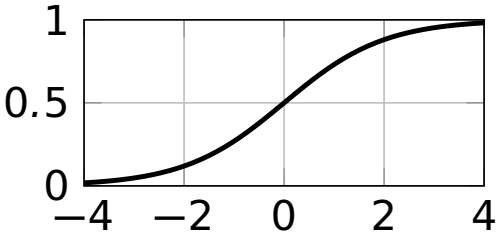
If the functions are linear, a sequence of functions doesn't give us anything – we can express two matrix multiplications as a single matrix. However, with nonlinear functions, a sequence of functions may be more complicated than a single function. The simplest way to do this is to first use a linear map, and then apply a nonlinear function to each dimension.

The benefit is that we can approximate a complicated function using a sequence of simple functions. We can make the approximation more accurate by having a longer sequence, or by increasing the dimensionality of the intermdiate representation $f_1(x)$. (The dimensionalities of the input x and output $f_2(f_1(x))$ are fixed by the data.)

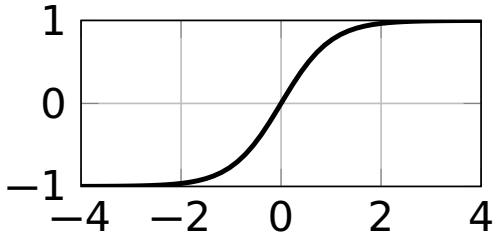
In practice, there will usually also be a "bias" term: $f(x) = g(Ax + b)$. (In strict mathematical terminology, $Ax + b$ would be called "affine", rather than "linear", but in machine learning, many authors use the term "linear".)

Nonlinear Activation Functions

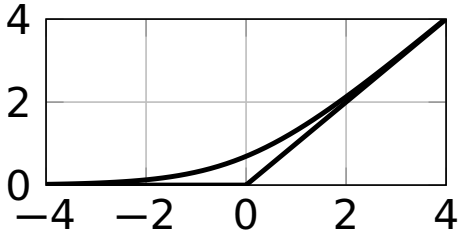
- $\frac{1}{1+e^{-x}}$ “sigmoid” (cf. logistic regression)
- $\frac{1-e^{-2x}}{1+e^{-2x}}$ “tanh”
- $\max\{x, 0\}$ “rectified linear”
- $\log(1+e^{x})$ “softplus”



We came across the sigmoid function when looking at logistic regression. It is also called the logistic function.

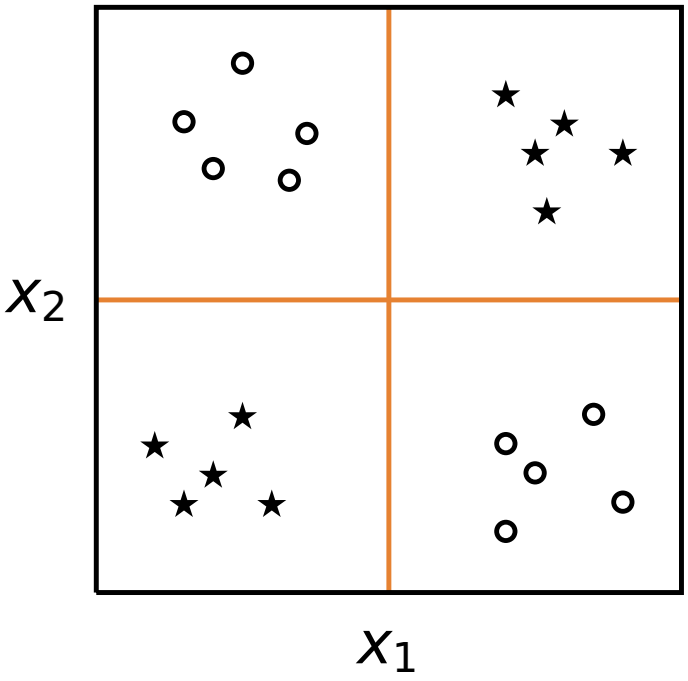


The tanh function (pronounced “tanch”, short for “hyperbolic tangent”), is important mathematically, but for reasons irrelevant here. It’s a rescaled sigmoid function, bounded between -1 and 1, and shrunk in the x direction.



The rectified linear unit is possibly the simplest nonlinearity, and fast to calculate. It isn’t differentiable at 0, and to avoid this, we can use the softplus function, which is smoothed out around 0.

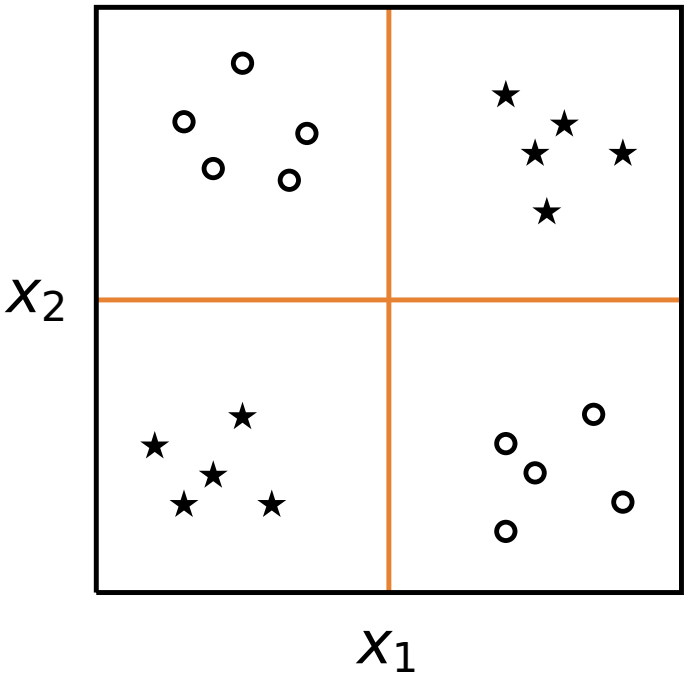
Nonlinear Decision Boundaries



Can be done with a decision tree

We have seen how decision trees allow us to learn nonlinear decision boundaries.

Nonlinear Decision Boundaries



Rectified linear units:

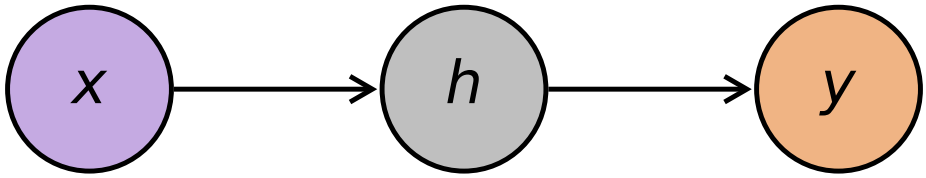
$$
\begin{aligned}
& r(\ \ x_1 + x_2 - 2) \\
+ & r(-x_1 - x_2 + 2) \\
- & r(\ \ x_1 - x_2) \\
- & r(-x_1 + x_2)
\end{aligned}
$$

A feedforward net can also learn a nonlinear decision boundary.

Here, r is the rectified linear function. We linearly map the input to a 4-dimensional vector, and then apply r componentwise. We then linearly map this vector to a single number – if it's above 0, we choose ⋆, and if it's below 0, we choose ∘.

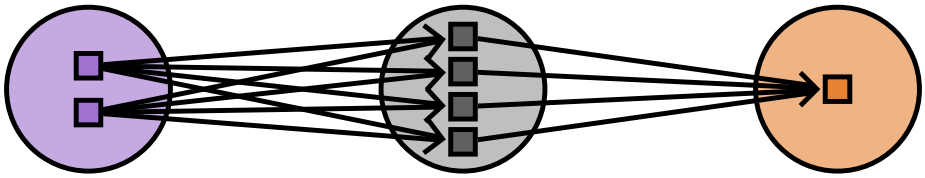
This is less interpretable than the decision tree, but neural networks give us a wider class of models.

Feedforward Networks



We can draw each vector in a neural net as a node in a graph.

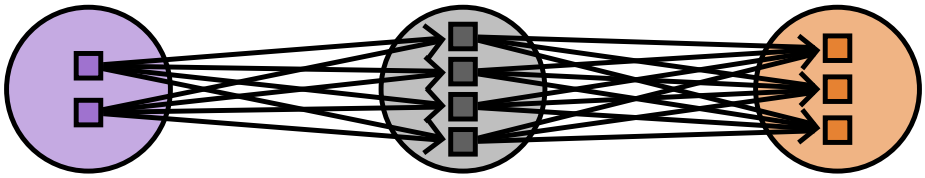
Feedforward Networks



We can also draw individual units (individual dimensions).

In the example from the previous slide, we had two input units, four hidden units, and one output unit (to decide between the two classes).

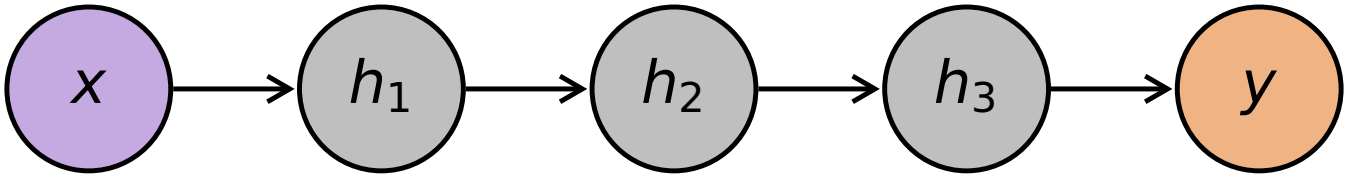
Feedforward Networks



Multiple classes: “softmax”
(multiclass logistic regression)

For multiple classes, we can use a softmax layer, which has one unit for each class. Mathematically, it’s the same as multiclass logistic regression.

“Deep” Feedforward Networks



A “deep” network is a network with many layers.

The choice of the term “deep” was good for publicity. The word has connotations of being “meaningful” or “serious”, and it sounds much more exciting than “function approximation parametrised by the composition of a sequence of simple functions”

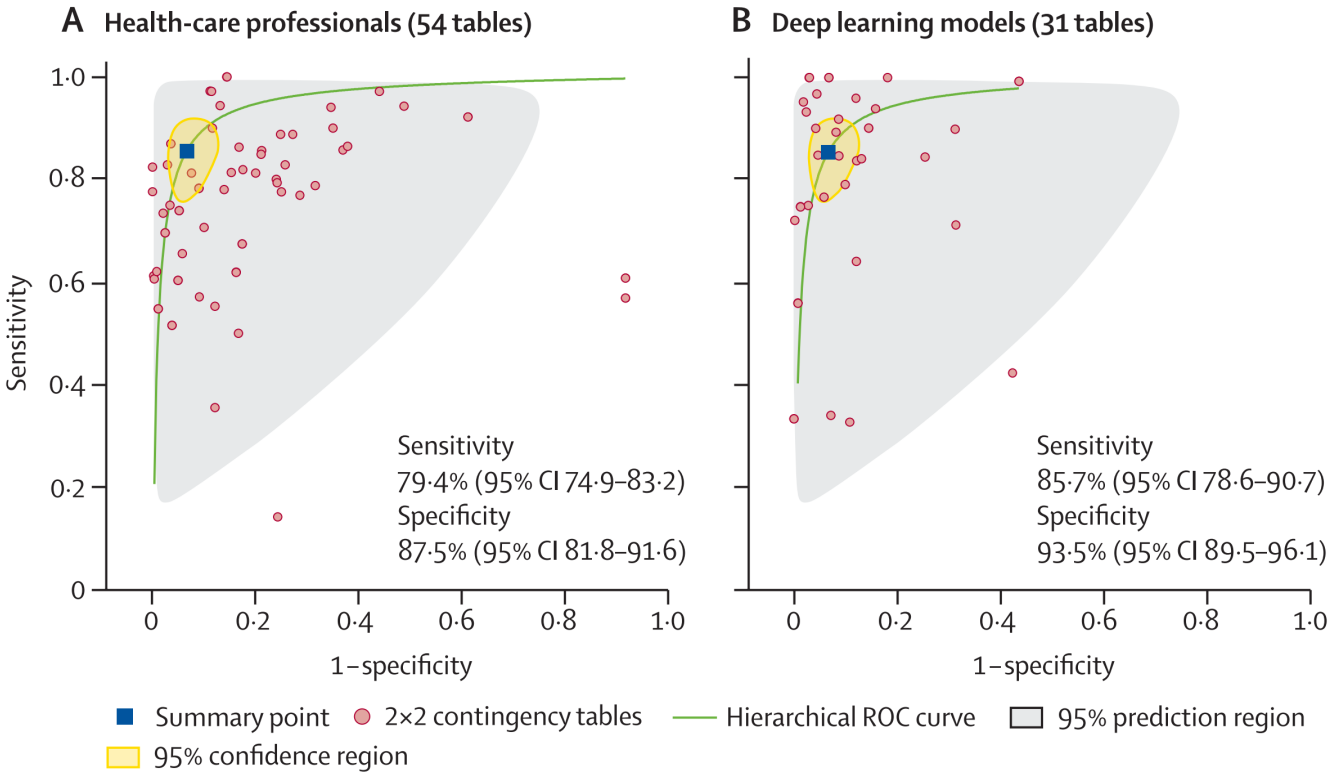
AlphaGo

AlphaGo, a Go-playing program based on deep learning, has consistently beaten the world's best Go players.

It has not been expected that a program could reach this level of performance – at least, based on traditional approaches to AI.

Image from: `https://deepmind.com/alphago-china`

Diagnosis from medical imaging



In a more practical setting, deep learning has also been applied to medical diagnosis.

This image comes from a metastudy, comparing deep learning models and medical professionals, tested on the same medical images. Each dot is one study, and the best performance would be in the top left corner. We can see that overall, the deep learning systems are performing about as well as the professionals.

Liu et al. (2019) "A comparison of deep learning performance against health-care professionals in detecting diseases from medical imaging: a systematic review and meta-analysis" https://www.thelancet.com/journals/landig/article/PIIS2589-7500(19)30123-2/fulltext

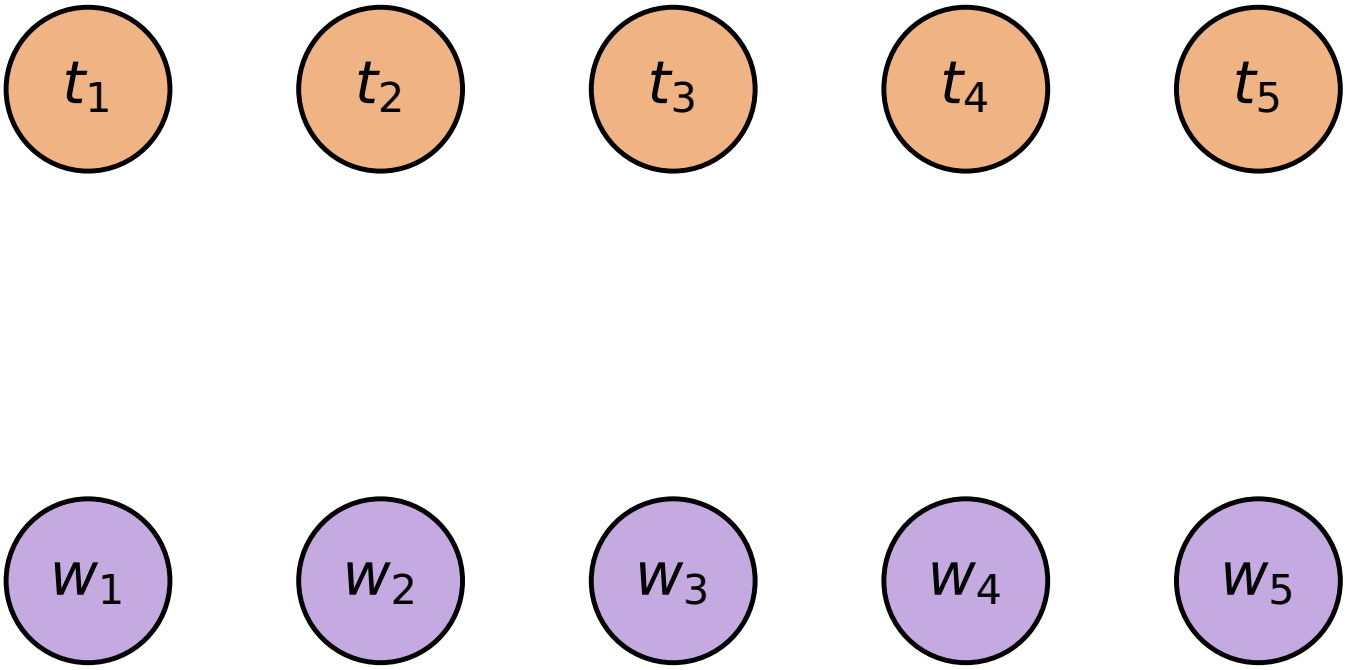
Sequence Labelling

article *noun* *verb* *article* *noun*

Every picture tells a story

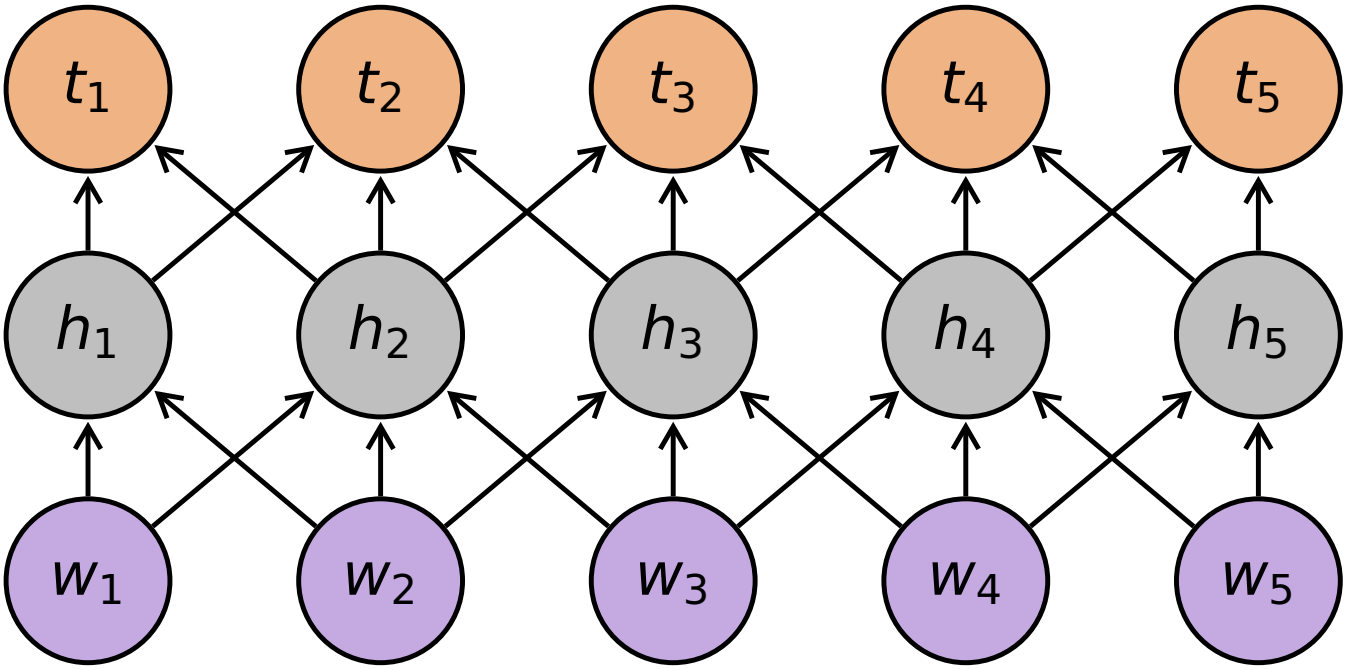
In part-of-speech tagging, the task is to label each word in a sentence with the correct part of speech.

Sequence Labelling



More generally, “sequence labelling” refers to tasks where we have one output t_i for each token w_i.

Convolutional Neural Net

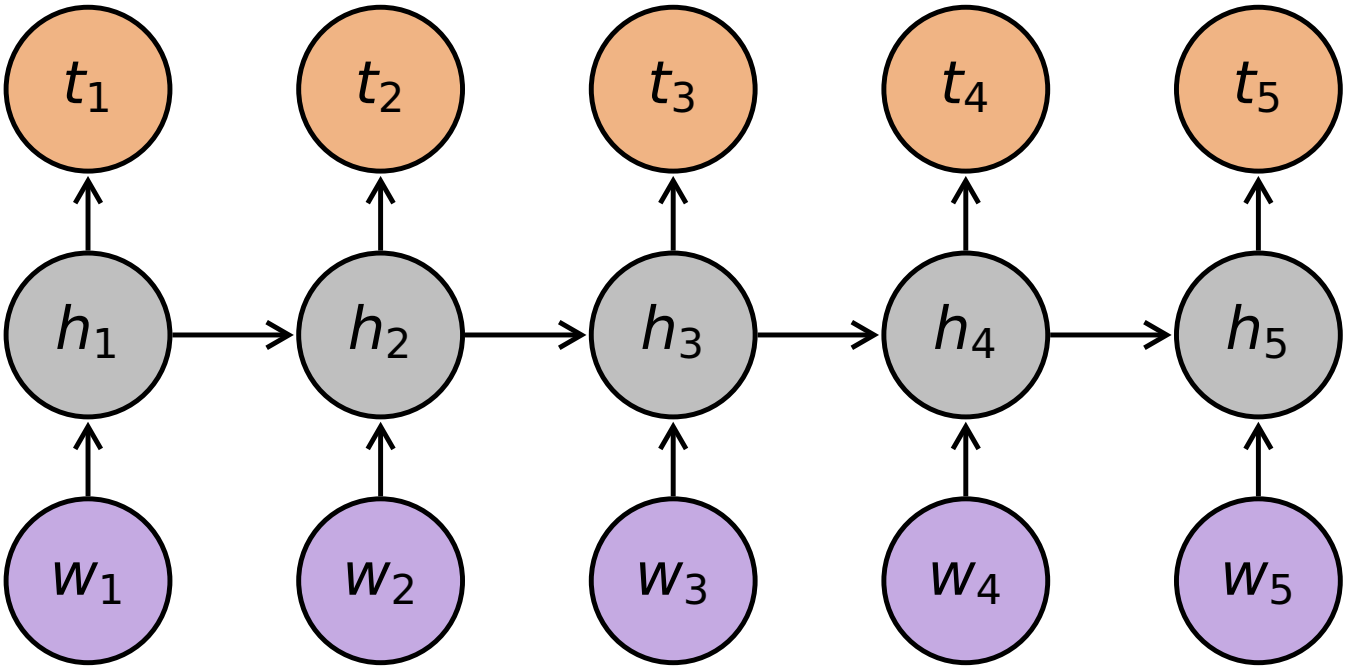


In a convolutional neural net (CNN), each hidden vector (and each output) is a function of a window of vectors – in this diagram, a window of one token either side.

The same function is used at each token – e.g. $h_2 = f(w_1, w_2, w_3)$ is the same function as $h_3 = f(w_2, w_3, w_4)$. Applying the same function across different windows is called a "convolution".

More precisely, the input vectors are concatenated – given three vectors w_1, w_2, w_3, each with N dimensions, we can view them together as one vector w_1^3 with $3N$ dimensions. We can then apply a normal feedforward layer – e.g. $h_2 = g(Aw_1^3 + b)$, for a matrix A, vector b, and nonlinearity g.

Recurrent Neural Net



One limitation of a CNN is that each output is a function of a limited window of words (on the previous slide, two words either side). However, for some tasks, we may need to take into account a larger context (for example, long-distance syntactic dependencies).

In a recurrent neural network (RNN), we have a hidden state which is dependent on the current token and the previous hidden state. This means that each prediction is dependent on the current token and all previous tokens. For example, t_5 depends on all input tokens – in the CNN on the previous slide, t_5 only depended on w_3 to w_5.

In a "vanilla" RNN, we concatenate w_i and h_{i-1}, and use a normal feedforward layer. The same function is used at each token.

Training a Network

- Loss function: $\mathcal{L}(\hat{y}, y)$
- Gradient wrt parameters: $\frac{d}{d\theta}\left(\mathcal{L}(\hat{y}, y)\right)$
- Update: $\theta \leftarrow \theta - \alpha \frac{d}{d\theta}\left(\mathcal{L}(\hat{y}, y)\right)$

The most common way to train a neural net is using gradient descent, which we saw in a previous lecture.

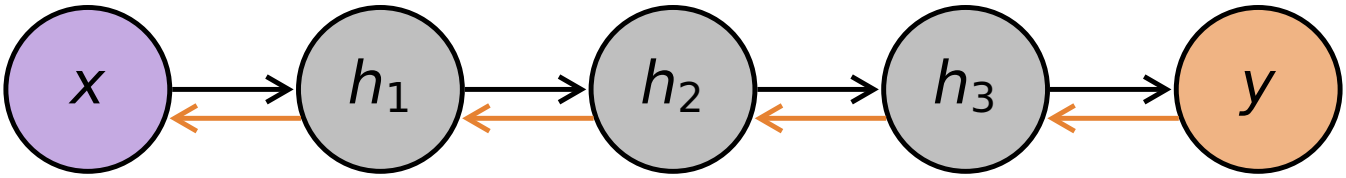
Backpropagation

- Chain rule: $\frac{d\mathcal{L}}{d\theta} = \frac{d\mathcal{L}}{du}\frac{du}{d\theta}$
- Backprop: efficient chain rule

Because a neural net is composed of many layers, the gradients can be calculated using the chain rule. An efficient algorithm to apply the chain rule is backpropagation, which we will see in more detail next lecture.

Backpropagation

Forward pass



Backward pass
(calculate gradients with chain rule)

To train a network, we apply the network forwards to get its predictions, and then use backpropagation to get the gradients.

Training Hyperparameters

- Large learning rate:
  - Faster training
- Small learning rate:
  - More stable training

We saw in the practicals how the learning rate is an important hyperparameter for gradient descent.

We will now look at some more hyperparameters.

Gradient Descent

- Loss per datapoint: $\mathcal{L}(\hat{y}_i, y_i)$
- Total training loss: $\sum_{i=1}^{N} \mathcal{L}(\hat{y}_i, y_i)$
- Ideal gradient: $\frac{d}{d\theta}\left(\sum_{i=1}^{N} \mathcal{L}(\hat{y}_i, y_i)\right)$

Ideally, the gradients would be calculated according to the total loss across the whole training set.

However, this is computationally expensive for large datasets.

Stochastic Gradient Descent

- Ideal gradient: $\sum_{i=1}^{N} \frac{d}{d\theta} \mathcal{L}(\hat{y}_i, y_i)$
- Stochastic gradient: $\frac{d}{d\theta} \mathcal{L}(\hat{y}_i, y_i)$ for $i = 1, 2, 3, \ldots$
- Minibatch gradient: $\sum_{i=j+1}^{j+b} \frac{d}{d\theta} \mathcal{L}(\hat{y}_i, y_i)$ for $j = 0, b, 2b, \ldots$

Rather than waiting until we've covered the entire training set before making an update to the model parameters, stochastic gradient descent (SGD) makes an update for each datapoint.

The downside of this is that the gradients can vary substantially from one datapoint to the next.

A middle ground is to make an update for each "minibatch" of datapoints. The batch size (b above) is an additional hyperparameter.

Training Hyperparameters

- Small batch size, large learning rate:
  - Faster training
- Large batch size, small learning rate:
  - More stable training

Varying the batch size has a similar effect to varying the learning rate.

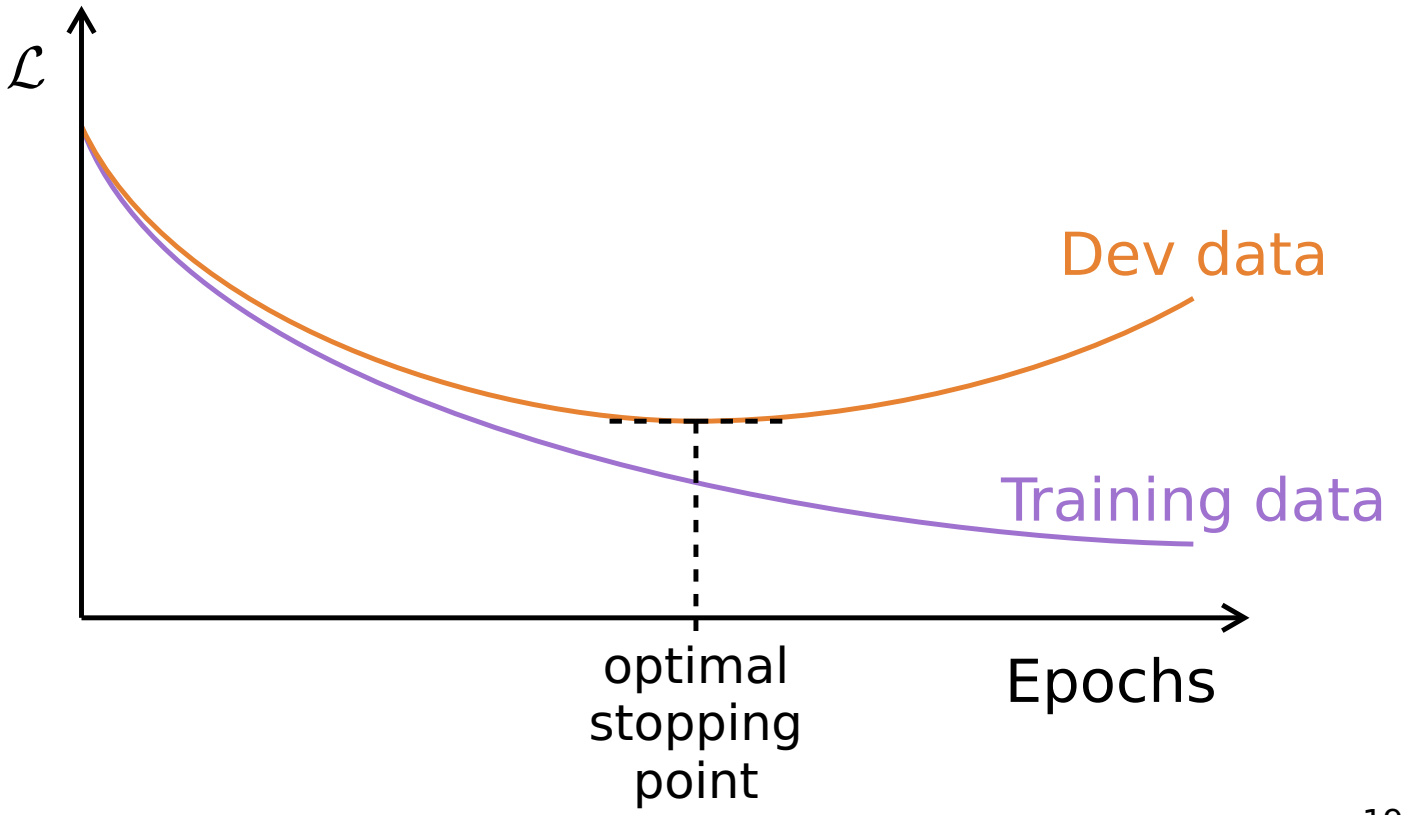
Overfitting

- Neural nets have many parameters
- Easy to overfit
- Some solutions:
  - Early stopping
  - Regularisation
  - Dropout

Overfitting is a general problem, not just for neural nets. However, neural nets can have so many parameters that it becomes a serious problem.

Early stopping and regularisation are also applicable to non-neural models, but dropout is specific to neural nets.

Early stopping



If a model overfits, the loss on the training set will decrease, but the loss on a held-out test set will increase.

The number of epochs (number of passes over the training data) can be treated as an additional hyperparameter, to be tuned on a development set.

Benefit: easy to implement.

Drawback: cannot benefit from further computation time.

Regularisation

- Penalise “bad” parameters:

  $\mathcal{L} = \mathcal{L}_{\text{err}}(\hat{y}_i, y_i) + \mathcal{L}_{\text{reg}}(\theta)$

- For example:

  $\mathcal{L}_1(\theta) = \sum_i |\theta_i|$

  $\mathcal{L}_2(\theta) = \sum_i |\theta_i|^2$

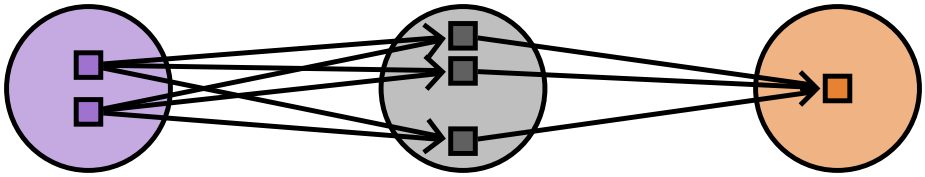
Rather than stopping training just before we overfit, we can try to avoid the model overfitting in the first place.

Regularisation does this by adding an additional term to the loss function, which penalises overfitted parameters.

To design a regularisation term, we need to understand what overfitted parameters look like. A simple approach is to assume that very large values may be overfitted, since the model may be relying on these too much. L1 regularisation penalises the absolute value, and L2 regularisation penalises the square value.

Using regularisation adds hyperparameters, e.g. $\mathcal{L} = \mathcal{L}_{\text{err}} + \lambda_1 \mathcal{L}_1 + \lambda_2 \mathcal{L}_2$

Dropout



- Less dependent on specific units
- More robust to noise

With dropout, each time we apply the network during training, we randomly switch off some units.

The probability of switching off each unit can be treated as an additional hyperparameter. Using a probability of 0.5 usually works well.

At test time, we can use the expected activation of each unit. For example, with a dropout probability of 0.5, we use all units but with half the activation.

The best way to apply dropout can depend on the architecture. Further reading: Gal and Ghahramani (2016) give a Bayesian analysis of dropout and apply it to RNNs. This is an example of how theory can take time to catch up with practice.

https://papers.nips.cc/paper/6241-a-theoretically-grounded-application-of-dropout-in-recurrent-neural-networks

What we've covered

- Neural nets, activation functions
- Architectures: CNNs, RNNs
- Training by gradient descent
- Early stopping, regularisation, dropout