

Feedforward Neural Networks: Revisiting Word Vectors and Text Classification

COM6513 Natural Language Processing

Nikos Aletras

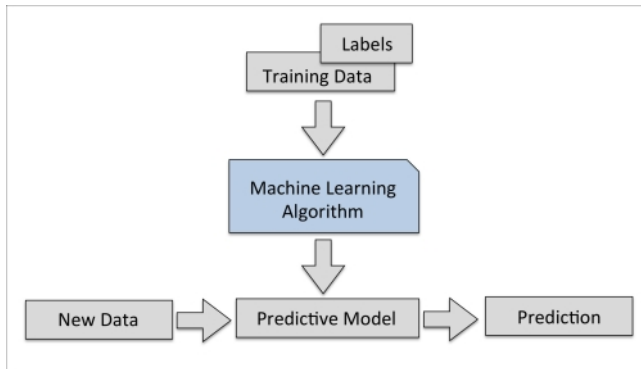
`n.aletras@sheffield.ac.uk`

Computer Science Department

Week 6
Spring 2023



In lecture 2...



Supervised ML

In lecture 2...

- **Machine Learning Algorithm:** Logistic Regression
- Binary and Multi-class

Logistic Regression recap

- Compute the dot product z between the input vector \mathbf{x} and the weight vector \mathbf{w} , and add a bias term b (often ignored):

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

Logistic Regression recap

- Compute the dot product z between the input vector \mathbf{x} and the weight vector \mathbf{w} , and add a bias term b (often ignored):

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

- Compute the probability of the positive class using the sigmoid function $\sigma(\cdot)$:

$$P(y = 1|\mathbf{x}; \mathbf{w}) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

Logistic Regression recap

- Compute the dot product z between the input vector \mathbf{x} and the weight vector \mathbf{w} , and add a bias term b (often ignored):

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

- Compute the probability of the positive class using the sigmoid function $\sigma(\cdot)$:

$$P(y = 1|\mathbf{x}; \mathbf{w}) = \sigma(z) = \frac{1}{1 + \exp(-z)}$$

- Predict the class with the highest probability:

$$\hat{y} := \begin{cases} 0 & \text{if } P(y = 1|\mathbf{x}; \mathbf{w}) < 0.5 \\ 1 & \text{otherwise} \end{cases}$$

Logistic Regression recap

- Extend to multi-class by introducing **weights for each class** and use **softmax** instead of sigmoid

Logistic Regression recap

- Extend to multi-class by introducing **weights for each class** and use **softmax** instead of sigmoid
- Learn the weights by minimising the **cross-entropy loss** using **Stochastic Gradient Descent**

Logistic Regression recap

- Extend to multi-class by introducing **weights for each class** and use **softmax** instead of sigmoid
- Learn the weights by minimising the **cross-entropy loss** using **Stochastic Gradient Descent**
- LR directly maps input to output and only captures linear relationships in the data

In this lecture...

- **Feedforward neural networks** or deep feedforward networks or multilayer perceptrons

In this lecture...

- **Feedforward neural networks** or deep feedforward networks or multilayer perceptrons
- Pass input through a series of intermediate computations (**hidden layers**) to capture **non-linear relationships** a.k.a. **deep learning**

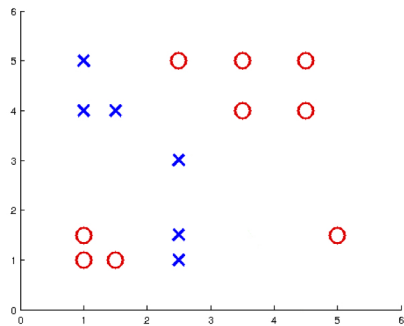
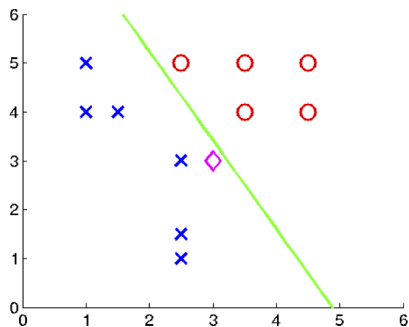
In this lecture...

- **Feedforward neural networks** or deep feedforward networks or multilayer perceptrons
- Pass input through a series of intermediate computations (**hidden layers**) to capture **non-linear relationships** a.k.a. **deep learning**
- Train with **SGD** and **Backpropagation** (for computing the gradients)

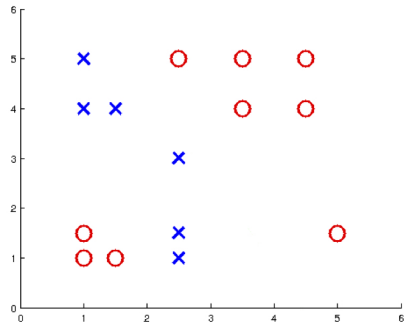
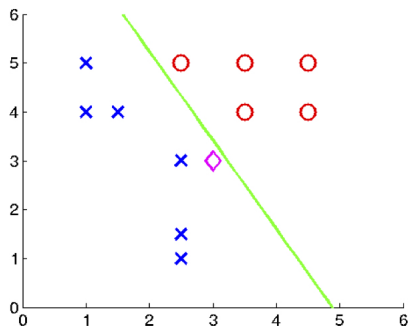
In this lecture...

- **Feedforward neural networks** or deep feedforward networks or multilayer perceptrons
- Pass input through a series of intermediate computations (**hidden layers**) to capture **non-linear relationships** a.k.a. **deep learning**
- Train with **SGD** and **Backpropagation** (for computing the gradients)
- NLP applications: word vectors and text classification

Limitations of linear models



Limitations of linear models



The righthand dataset is not linearly separable and cannot be learned with a linear model.

Feedforward Neural Network

- A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ between an input \mathbf{x} and output \mathbf{y} given parameters \mathbf{w} .

Feedforward Neural Network

- A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ between an input \mathbf{x} and output \mathbf{y} given parameters \mathbf{w} .
- Feedforward nets compose together many different **functions** connected in a **chain**: $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$

Feedforward Neural Network

- A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ between an input \mathbf{x} and output \mathbf{y} given parameters \mathbf{w} .
- Feedforward nets compose together many different **functions** connected in a **chain**: $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$
- f_1 is the first **hidden layer** of the model, f_2 the second and so on. Number of hidden layers denote the **depth** of the model.

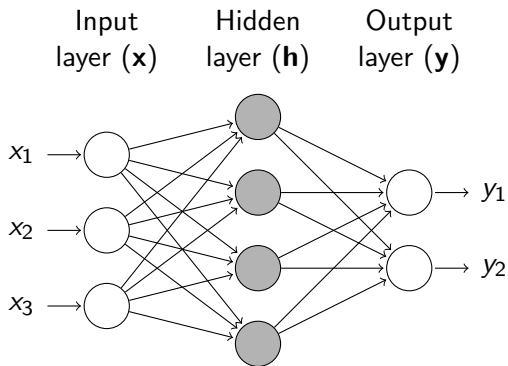
Feedforward Neural Network

- A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ between an input \mathbf{x} and output \mathbf{y} given parameters \mathbf{w} .
- Feedforward nets compose together many different **functions** connected in a **chain**: $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$
- f_1 is the first **hidden layer** of the model, f_2 the second and so on. Number of hidden layers denote the **depth** of the model.
- **Input** denote the input layer

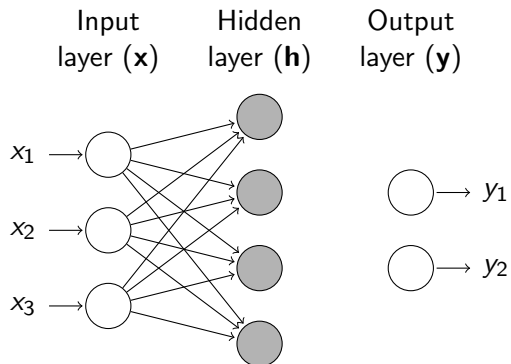
Feedforward Neural Network

- A feedforward network defines a mapping $\mathbf{y} = f(\mathbf{x}; \mathbf{w})$ between an input \mathbf{x} and output \mathbf{y} given parameters \mathbf{w} .
- Feedforward nets compose together many different **functions** connected in a **chain**: $f(\mathbf{x}) = f_3(f_2(f_1(\mathbf{x})))$
- f_1 is the first **hidden layer** of the model, f_2 the second and so on. Number of hidden layers denote the **depth** of the model.
- **Input** denote the input layer
- The final layer to obtain the prediction is called the **output** layer (e.g. sigmoid, softmax)

Feedforward Neural Network

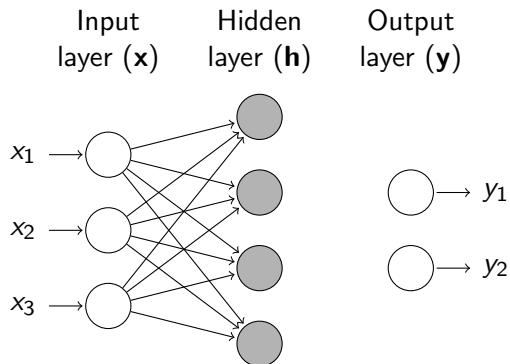


Feedforward Neural Network



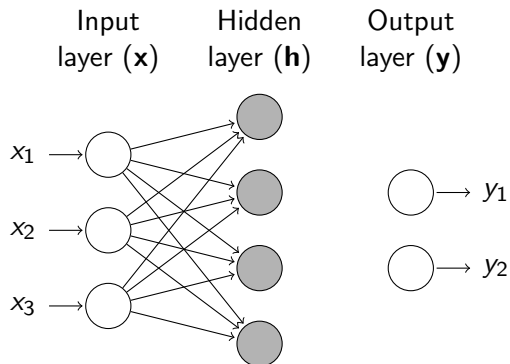
$$\mathbf{x} \in \mathcal{R}^d, d = 3$$

Feedforward Neural Network



$$\mathbf{x} \in \mathcal{R}^d, d = 3$$
$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$

Feedforward Neural Network

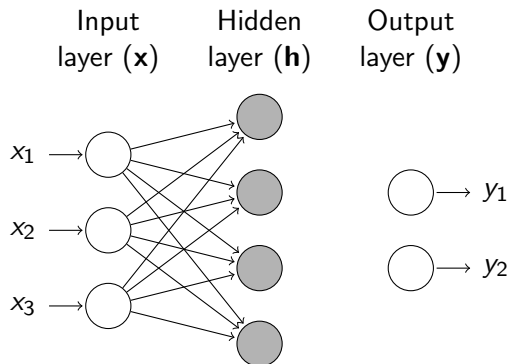


$$\mathbf{x} \in \mathcal{R}^d, d = 3$$

$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$

$$\mathbf{h} \in \mathcal{R}^h, h =$$

Feedforward Neural Network



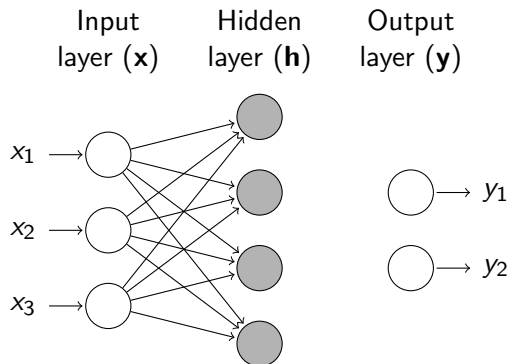
$$\mathbf{x} \in \mathcal{R}^d, d = 3$$

$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$

$$\mathbf{h} \in \mathcal{R}^h, h = 4$$

$$\mathbf{W}_h \in$$

Feedforward Neural Network



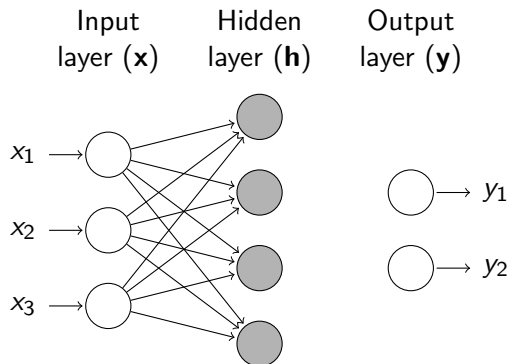
$$\mathbf{x} \in \mathcal{R}^d, d = 3$$

$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$

$$\mathbf{h} \in \mathcal{R}^h, h = 4$$

$$\mathbf{W}_h \in \mathcal{R}^{d \times h}$$

Feedforward Neural Network

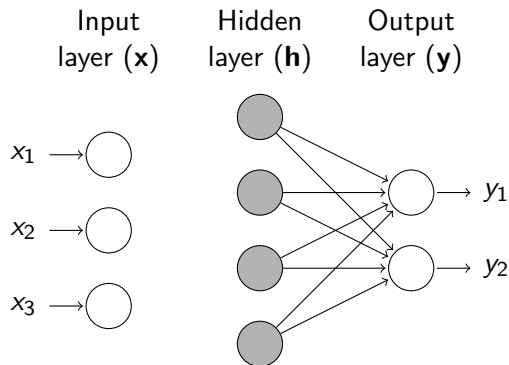


$$\begin{aligned}\mathbf{x} &\in \mathcal{R}^d, d = 3 \\ \mathbf{h} &= g(\mathbf{x}^T \mathbf{W}_h) \\ \mathbf{h} &\in \mathcal{R}^h, h = 4 \\ \mathbf{W}_h &\in \mathcal{R}^{d \times h}\end{aligned}$$

Extended to deeper architectures:

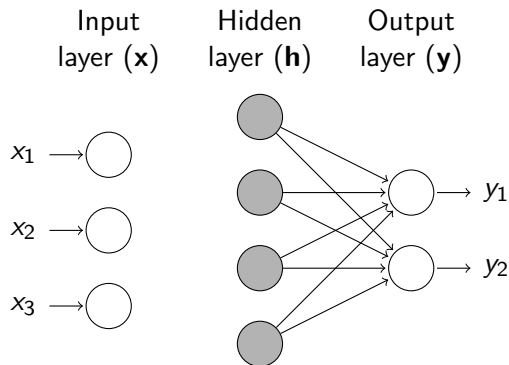
$$\mathbf{h}_i = g(\mathbf{h}_{i-1}^T \mathbf{W}_{h_i})$$

Feedforward Neural Network



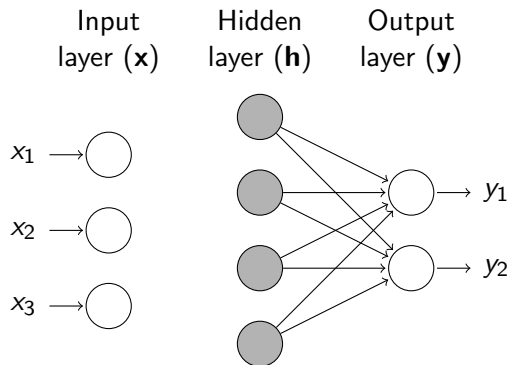
$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$

Feedforward Neural Network



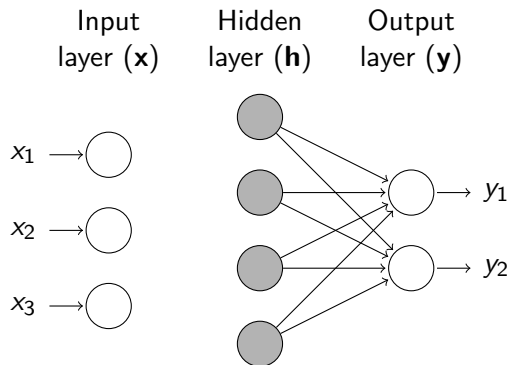
$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$
$$\mathbf{y} = \text{softmax}(\mathbf{h}^T \mathbf{W}_o)$$
$$\mathbf{W}_o \in$$

Feedforward Neural Network



$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$
$$\mathbf{y} = \text{softmax}(\mathbf{h}^T \mathbf{W}_o)$$
$$\mathbf{W}_o \in \mathcal{R}^{h \times y}$$

Feedforward Neural Network



$$\mathbf{h} = g(\mathbf{x}^T \mathbf{W}_h)$$
$$\mathbf{y} = \text{softmax}(\mathbf{h}^T \mathbf{W}_o)$$
$$\mathbf{W}_o \in \mathcal{R}^{h \times y}$$

But what is $g(\cdot)$?

Activation Functions

- Applied on **hidden units**: elements of h

Activation Functions

- Applied on **hidden units**: elements of h
- **Sigmoid**:

$$g(z) = \sigma(z)$$

Activation Functions

- Applied on **hidden units**: elements of h
- **Sigmoid**:

$$g(z) = \sigma(z)$$

- **Hyperbolic Tangent**:

$$g(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

Activation Functions

- Applied on **hidden units**: elements of h

- **Sigmoid**:

$$g(z) = \sigma(z)$$

- **Hyperbolic Tangent**:

$$g(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

- **Rectified Linear Unit (ReLU)**:

$$g(z) = \max(0, z)$$

Activation Functions

- Applied on **hidden units**: elements of h

- **Sigmoid**:

$$g(z) = \sigma(z)$$

- **Hyperbolic Tangent**:

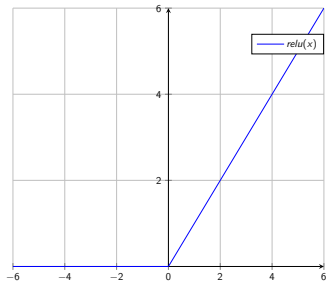
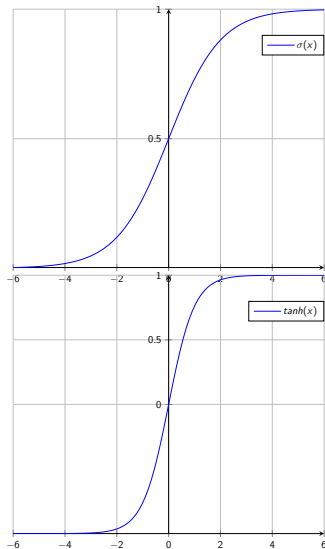
$$g(z) = \tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

- **Rectified Linear Unit (ReLU)**:

$$g(z) = \max(0, z)$$

- And many more...

Activation Functions



Training: Stochastic Gradient Descent (SGD) recap

Input: $D_{train} = \{(x_1, y_1) \dots (x_M, y_M)\}$, $D_{val} = \{(x_1, y_1) \dots (x_D, y_D)\}$,

learning rate η , epochs e , tolerance t

initialize \mathbf{w} with zeros

for each epoch e do

randomise order in D_{train}

for each (x_i, y_i) in D_{train} do

update $\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}; x_i, y_i)$

monitor training and validation loss

if *previous validation loss – current validation loss; smaller than t*

break

return \mathbf{w}

Training: Stochastic Gradient Descent (SGD) recap

Input: $D_{train} = \{(x_1, y_1) \dots (x_M, y_M)\}$, $D_{val} = \{(x_1, y_1) \dots (x_D, y_D)\}$,
learning rate η , epochs e , tolerance t
initialize \mathbf{w} with zeros
for each epoch e do
 randomise order in D_{train}
 for each (x_i, y_i) in D_{train} do
 update $\mathbf{w} = \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}; x_i, y_i)$
 monitor training and validation loss
 if *previous validation loss* — *current validation loss*; *smaller than t*
 break
return \mathbf{w}

How to compute the gradient for the weights of the hidden layers?

Training: Backpropagation Algorithm

- **Forward Pass:** Compute and store all the output values of all the hidden units (for each hidden layer) and the output layer

Training: Backpropagation Algorithm

- **Forward Pass:** Compute and store all the output values of all the hidden units (for each hidden layer) and the output layer
- **Backward Pass:** Compute the gradients for the output and hidden layers with respect to the cost function L and update the weights for each layer

Training: SGD and Backpropagation

Input: $D_{train} = \{(x_1, y_1) \dots (x_M, y_M)\}$, $D_{val} = \{(x_1, y_1) \dots (x_D, y_D)\}$,
learning rate η , epochs e , tolerance t
initialise $W_i \in W = \{W_1, \dots, W_l\}$ for each layer (small random values)
for each epoch e do
 randomise order in D_{train}
 for each (x_i, y_i) in D_{train} do
 layer_outputs = *forward_pass* $((x_i, y_i), W)$
 $W =$ *backward_pass* $((x_i, y_i), W, L, \text{layer_outputs})$
 monitor training and validation loss
 if *prev val loss* – *current val loss*; *smaller than t* : **break**
return W

Forward Pass

Propagate the inputs forward to compute the outputs for all layers:

$$\mathbf{h}_0 \leftarrow \mathbf{x} \text{ (input layer)}$$

Forward Pass

Propagate the inputs forward to compute the outputs for all layers:

```
 $h_0 \leftarrow x$  (input layer)  
for layer  $k = 1, \dots, l$  do
```

Forward Pass

Propagate the inputs forward to compute the outputs for all layers:

```
 $\mathbf{h}_0 \leftarrow \mathbf{x}$  (input layer)  
for layer  $k = 1, \dots, l$  do  
     $\mathbf{z}_k \leftarrow \mathbf{W}_k \mathbf{h}_{k-1}$   
     $\mathbf{h}_k \leftarrow g(\mathbf{z}_k)$   
end for
```

Forward Pass

Propagate the inputs forward to compute the outputs for all layers:

$\mathbf{h}_0 \leftarrow \mathbf{x}$ (input layer)

for layer $k = 1, \dots, l$ **do**

$\mathbf{z}_k \leftarrow \mathbf{W}_k \mathbf{h}_{k-1}$

$\mathbf{h}_k \leftarrow g(\mathbf{z})$

end for

Get prediction $\hat{\mathbf{y}} = \mathbf{h}_l$

Compute cross-entropy loss $L(\hat{\mathbf{y}}, \mathbf{y})$

return \mathbf{h}, \mathbf{z} for all layers

Backward Pass

Propagate the gradients backwards from the loss to the input layer (i.e. how each layer's output should change to reduce error):

Compute gradient on the output layer $\mathbf{g} \leftarrow \nabla_{\hat{y}} L$

Backward Pass

Propagate the gradients backwards from the loss to the input layer (i.e. how each layer's output should change to reduce error):

Compute gradient on the output layer $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} L$

for layer $k = l, l - 1, \dots, 1$ **do**

Convert the gradient on the layer's output (\mathbf{h}) into a gradient before the activation function (\mathbf{z}):

$\mathbf{g} \leftarrow \nabla_{\mathbf{z}_k} L = \mathbf{g} \odot f'(\mathbf{z}_k)$ (\odot element-wise, $f'(\cdot)$ deriv.)

Backward Pass

Propagate the gradients backwards from the loss to the input layer (i.e. how each layer's output should change to reduce error):

Compute gradient on the output layer $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} L$

for layer $k = l, l - 1, \dots, 1$ **do**

Convert the gradient on the layer's output (\mathbf{h}) into a gradient before the activation function (\mathbf{z}):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{z}_k} L = \mathbf{g} \odot f'(\mathbf{z}_k) \quad (\odot \text{ element-wise, } f'(\cdot) \text{ deriv.})$$

Compute gradients on weights:

$$\nabla_{\mathbf{W}_k} L = \mathbf{g} \mathbf{h}_{k-1}$$

Backward Pass

Propagate the gradients backwards from the loss to the input layer (i.e. how each layer's output should change to reduce error):

Compute gradient on the output layer $\mathbf{g} \leftarrow \nabla_{\hat{y}} L$

for layer $k = l, l - 1, \dots, 1$ **do**

Convert the gradient on the layer's output (\mathbf{h}) into a gradient before the activation function (\mathbf{z}):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{z}_k} L = \mathbf{g} \odot f'(\mathbf{z}_k) \quad (\odot \text{ element-wise, } f'(\cdot) \text{ deriv.})$$

Compute gradients on weights:

$$\nabla_{W_k} L = \mathbf{g} h_{k-1}$$

Compute the gradients w.r.t. the next hidden layer:

$$\mathbf{g} \leftarrow \nabla_{h_{k-1}} L = \mathbf{g} W_k$$

Backward Pass

Propagate the gradients backwards from the loss to the input layer (i.e. how each layer's output should change to reduce error):

Compute gradient on the output layer $\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} L$

for layer $k = l, l - 1, \dots, 1$ **do**

Convert the gradient on the layer's output (\mathbf{h}) into a gradient before the activation function (\mathbf{z}):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{z}_k} L = \mathbf{g} \odot f'(\mathbf{z}_k) \quad (\odot \text{ element-wise, } f'(\cdot) \text{ deriv.})$$

Compute gradients on weights:

$$\nabla_{W_k} L = \mathbf{g} h_{k-1}$$

Compute the gradients w.r.t. the next hidden layer:

$$\mathbf{g} \leftarrow \nabla_{h_{k-1}} L = \mathbf{g} W_k$$

Update current weights:

$$W_k \leftarrow W_k - \eta \nabla_{W_k} L$$

end for

return W

Regularisation

- **L2-regularisation** in the weights of each layer (added in the loss function of each layer)

Regularisation

- **L2-regularisation** in the weights of each layer (added in the loss function of each layer)
- **Dropout**: randomly ignore a percentage (e.g. 20% or 50%) of layer outputs during training:

Regularisation

- **L2-regularisation** in the weights of each layer (added in the loss function of each layer)
- **Dropout**: randomly ignore a percentage (e.g. 20% or 50%) of layer outputs during training:
 - Apply a random binary mask after the activation function, i.e. elementwise multiplication with vector containing 0s in random positions

Design Choices

- How many layers?
- How many units per layer?
- What activation function(s)?

Design Choices

- How many layers?
- How many units per layer?
- What activation function(s)?
- Architecture engineering vs feature engineering
- Theory says that we can approximate any function with one hidden layer, practice says different architectures work well for different problems

Implementation tips

- Learning objective non-convex: initialisation matters
 - start with small non-zero values
 - random restarts to escape local optima
- Greater learning capacity makes overfitting more likely: regularise
- Many open libraries are available: PyTorch, Tensorflow, MxNet, Keras etc.

Applications: Word Vectors

- Lecture 1: word vectors by **counting** co-occurrences with context words

Applications: Word Vectors

- Lecture 1: word vectors by **counting** co-occurrences with context words
- Instead, use a feedforward network to **predict** a context word for a given word (and vice versa)

Applications: Word Vectors

- Lecture 1: word vectors by **counting** co-occurrences with context words
- Instead, use a feedforward network to **predict** a context word for a given word (and vice versa)
- **Word2Vec (Mikolov et al., 2013)** family, more recently supporting char n-grams (e.g. FastText)

- **Skip-gram model:** Given a word predict its context words

Word2Vec

- **Skip-gram model:** Given a word predict its context words
- **Continuous BOW (CBOW):** Given context words predict the current word

Word2Vec

- **Skip-gram model:** Given a word predict its context words
- **Continuous BOW (CBOW):** Given context words predict the current word
- **Input:** A word, represented as a one-hot vector over vocabulary (or the vocabulary index for memory efficiency!)

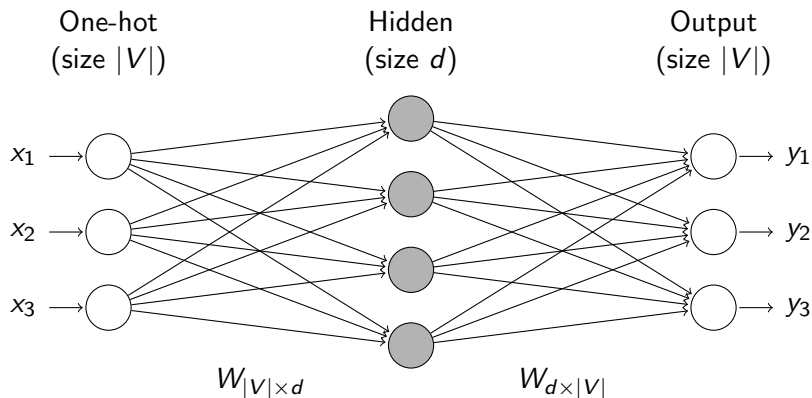
Word2Vec

- **Skip-gram model:** Given a word predict its context words
- **Continuous BOW (CBOW):** Given context words predict the current word
- **Input:** A word, represented as a one-hot vector over vocabulary (or the vocabulary index for memory efficiency!)
- **Hidden layer:** One hidden layer of size vocabulary \times hidden size (usually 300), linear activation function

Word2Vec

- **Skip-gram model:** Given a word predict its context words
- **Continuous BOW (CBOW):** Given context words predict the current word
- **Input:** A word, represented as a one-hot vector over vocabulary (or the vocabulary index for memory efficiency!)
- **Hidden layer:** One hidden layer of size vocabulary \times hidden size (usually 300), linear activation function
- **Output:** softmax over the vocabulary to predict the correct context/target words respectively

Word2Vec Architecture



Word2Vec

- Training data (x, y) can be obtained from large corpora

Word2Vec

- Training data (x, y) can be obtained from large corpora
- For Skip-gram:

the **cat** sat \rightarrow cat, {the, sat}

Word2Vec

- Training data (x, y) can be obtained from large corpora
- For Skip-gram:

the cat sat \rightarrow cat, {the, sat}

- For CBOW:

the cat sat \rightarrow {the, sat}, cat

Word2Vec

- Training data (x, y) can be obtained from large corpora
- For Skip-gram:

the **cat** sat \rightarrow cat, {the, sat}

- For CBOW:

the **cat** sat \rightarrow {the, sat}, cat

- **Vector** of a word $x_i = W_i$, from the network weights

Word2Vec

- Training data (x, y) can be obtained from large corpora
- For Skip-gram:

the **cat** sat \rightarrow cat, {the, sat}

- For CBOW:

the **cat** sat \rightarrow {the, sat}, cat

- **Vector** of a word $x_i = W_i$, from the network weights
- Evaluation: standard approaches for word representation (see Lecture 1)

Word2Vec

- Training data (x, y) can be obtained from large corpora
- For Skip-gram:

the **cat** sat \rightarrow cat, {the, sat}

- For CBOW:

the **cat** sat \rightarrow {the, sat}, cat

- **Vector** of a word $x_i = W_i$, from the network weights
- Evaluation: standard approaches for word representation (see Lecture 1)
- Pre-trained word embeddings are widely re-used in other NLP tasks, i.e. transfer learning (more in Lecture 10)

Word2Vec: Implementation Details

- Word2Vec is a huge neural network, to make the training feasible:

Word2Vec: Implementation Details

- Word2Vec is a huge neural network, to make the training feasible:
 - **Negative Sampling:** Update the weights for the positive word, plus the weights for a small number (5-20) other words that we want to output 0

Word2Vec: Implementation Details

- Word2Vec is a huge neural network, to make the training feasible:
 - **Negative Sampling:** Update the weights for the positive word, plus the weights for a small number (5-20) other words that we want to output 0
 - **Subsampling frequent words** to decrease the number of training examples

Applications: Text Classification

- **Approach 1:** Pass BOW vectors into a series of hidden layers (extending the LR model in Lecture 2)

Applications: Text Classification

- **Approach 1:** Pass BOW vectors into a series of hidden layers (extending the LR model in Lecture 2)
- **Approach 2:** Pass one-hot word vectors through an **embedding layer** to obtain embeddings for each word in a document which are subsequently **concatenated (or added/averaged) and passed through a series of hidden layers**

Applications: Text Classification

- **Approach 1:** Pass BOW vectors into a series of hidden layers (extending the LR model in Lecture 2)
- **Approach 2:** Pass one-hot word vectors through an **embedding layer** to obtain embeddings for each word in a document which are subsequently **concatenated (or added/averaged) and passed through a series of hidden layers**
- Approach 2 is more contemporary and usually the embedding layer is pre-trained (e.g. using Word2Vec) and is not updated during training

Bibliographhy

- Chapters 6-8 from Goodfellow et al.
- Sections 3-6 from Goldberg
- Tutorial on backprop by D. Stansbury

Coming up next..

- Information Extraction by Dr. Samuel Mensah