Advanced Topics in Machine Learning

Alejo Nevado-Holgado

Lecture 11 (NLP 3) - Classification and neural networks

V 0.6 (15 Feb 2020 - final version)

Feedback so far

- Define all the technical terms that you use (e.g. hyperparameters) [+1/ 0]
- Sometimes you talk too fast [+1/ 0]
- The speed/amount of material is good [+3/ 0]
- The spoken descriptions of the equations, and why they are used, are very useful [+1/ 0]

Course structure

- **Introduction:** What is NLP. Why it is hard. Why NNs work well ← Lecture 9 (NLP 1)
- **Word representation:** How to represent the meaning of individual words
  - Old technology: One-hot representations, synsets ← Lecture 9 (NLP 1)
  - Embeddings: First trick that boosted the performance of NNs in NLP ← Lecture 9 (NLP 1)
    - Word2vec: Single layer NN. CBOW and skip-gram ← Lecture 10 (NLP 2)
    - Co-occurrence matrices: Basic counts and SVD improvement ← Lecture 10 (NLP 2)
    - Glove: Combining word2vec and co-occurrence matrices idea ← Lecture 10 (NLP 2)
    - Evaluating performance of embeddings ← Lecture 10 (NLP 2)
- **Named Entity Recognition (NER):** How to find words of specific meaning within text
  - Multilayer NNs: Margin loss. Forward- and back-propagation ← Lecture 11 (NLP 3)
  - Better loss functions: margin loss, regularisation ← Lecture 11 (NLP 3)
  - Better initializations: uniform, xavier ← Lecture 11 (NLP 3)
  - Better optimizers: Adagrad, RMSprop, Adam... ← Lecture 11 (NLP 3)

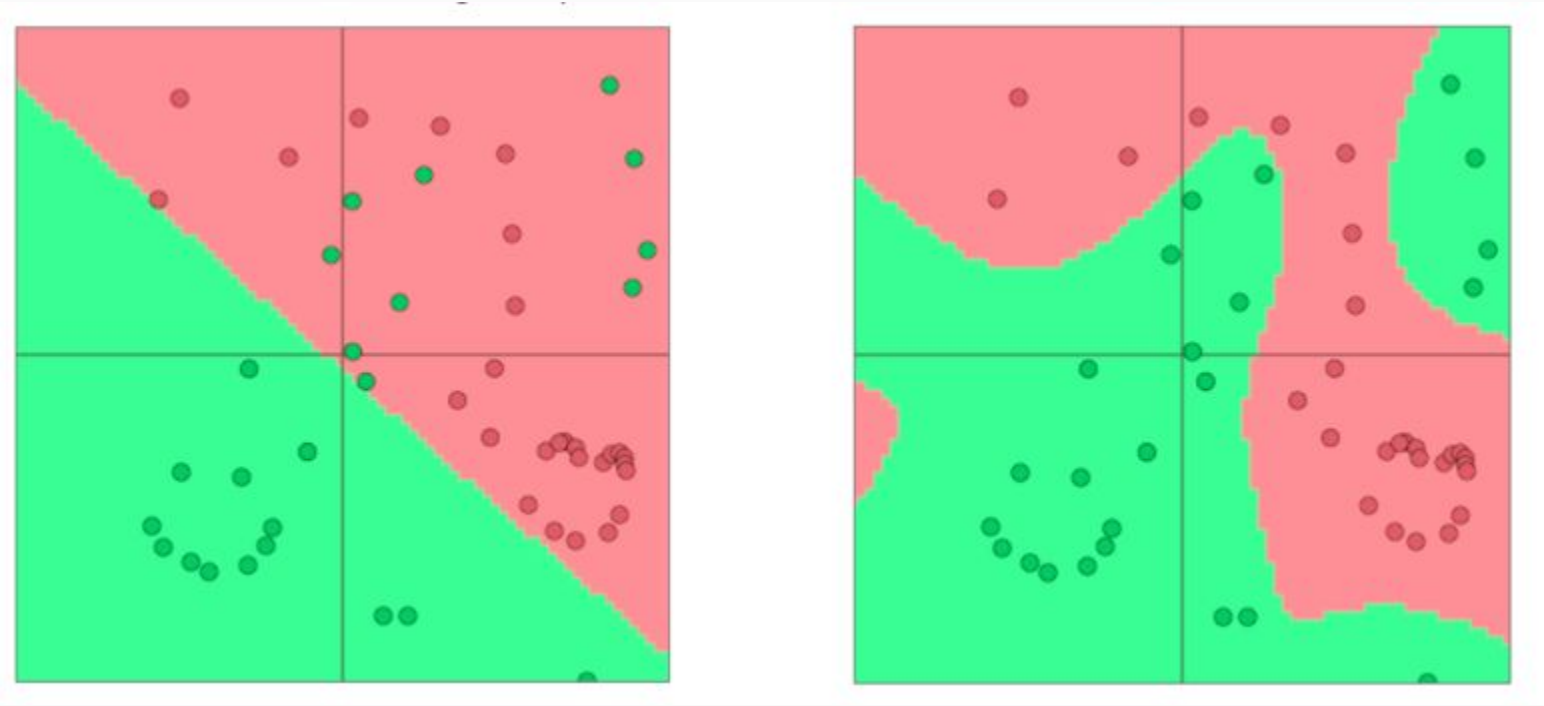
Course structure

- **Language modelling:** How to represent the meaning of full pieces of text
  - Old technology: N-grams ← Lecture 12 (NLP 4)
  - Recursive NNs language models (RNNs) ← Lecture 12 (NLP 4)
  - Evaluating performance of language models ← Lecture 12 (NLP 4)
  - Vanishing gradients: Problem. Gradient clipping ← Lecture 13 (NLP 5)
  - Improved RNNs: LSTM, GRU ← Lecture 13 (NLP 5)
- **Machine translation:** How to translate text
  - Old technology: Georgetown–IBM experiment and ALPAC report ← Lecture 16 (NLP 6)
  - Seq2seq: Greedy decoding, encoder-decoder, beam search ← Lecture 16 (NLP 6)
  - Attention: Simple attention, transformers, reformers ← Lecture 16 (NLP 6)
  - Evaluating performance: BLEU ← Lecture 16 (NLP 6)

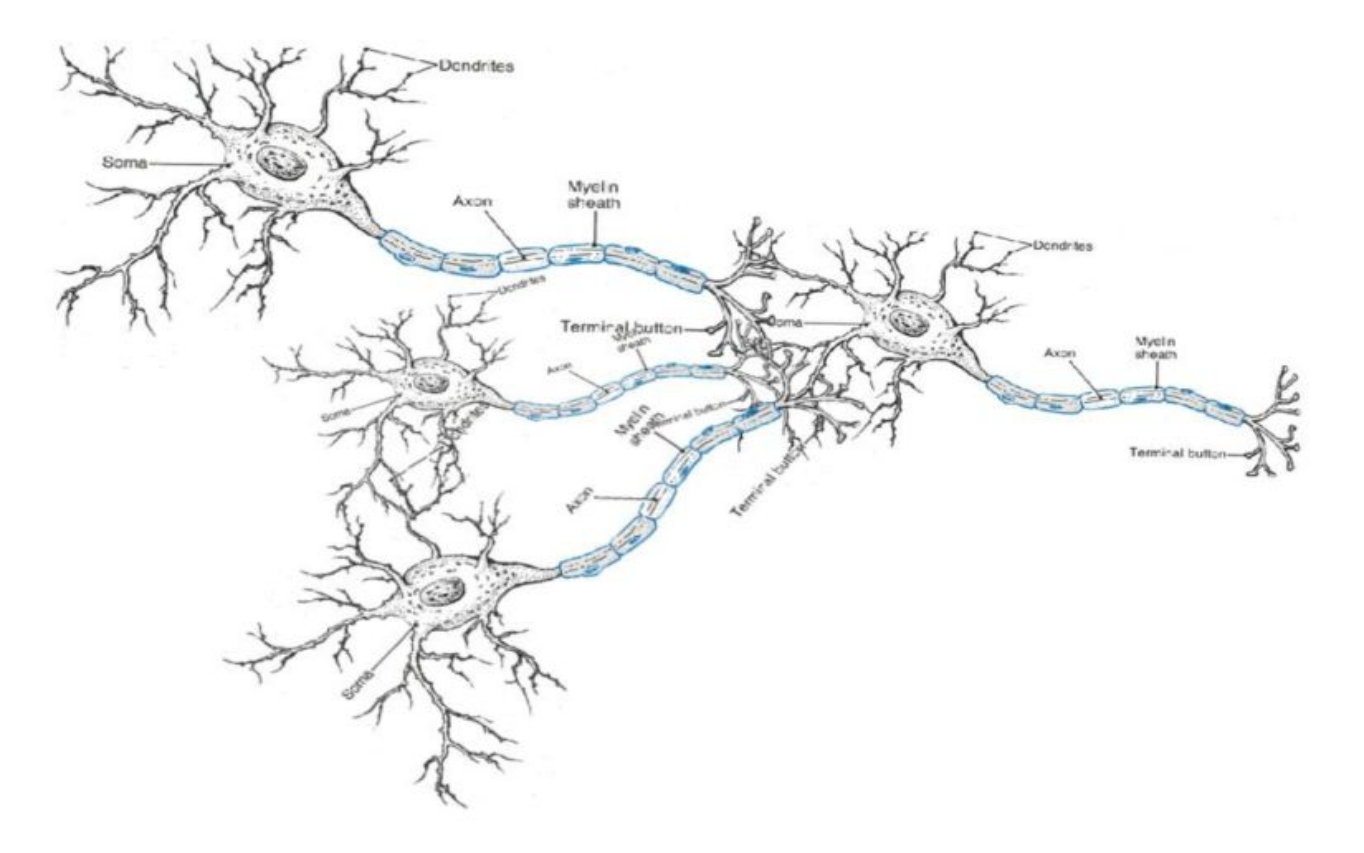
Neural networks

- Machine learning models can be divided into linear and nonlinear
- Linear: Rarely overfit. Better understood. They performance plateau
  - Linear statistical models: Very well understood. Use very extensively outside computer science and statistics (medicine, biology, psychology…). Has many flavours (mixed effects model, general linear model, generalized linear model…)
  - Others: linear discriminant analysis, Support vector machines, ...
- Nonlinear: Often overfit. Better performance when there is a lot of data. Long to train
  - Neural networks: Very flexible. An algorithm exists to calculate its gradient (backpropagation). Currently state of the art. They are black box
  - Others: Quadratic statistical models, splines, support vector machines with kernel trick, genetic algorithms...

Neural networks

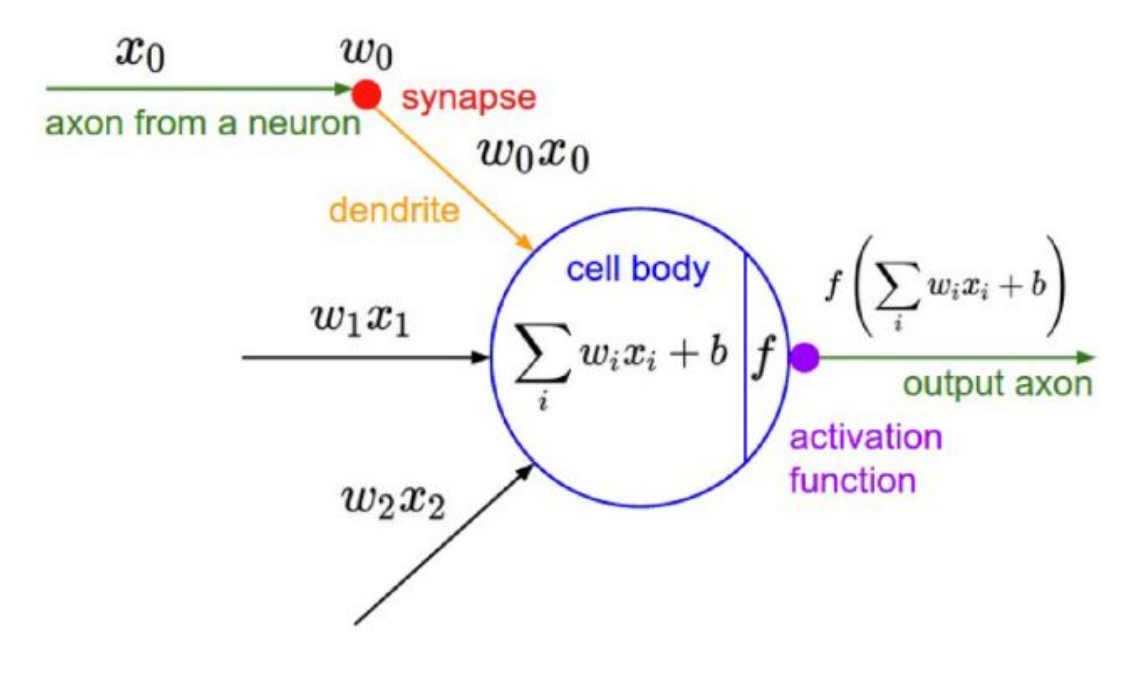


Neural networks



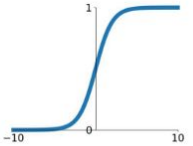
Neural networks

Mathematically, the model used by neural networks to represent each neuron is a “linear logistic regression” (if f = sigmoid)



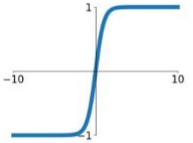
Sigmoid

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



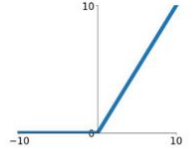
tanh

$\tanh(x)$



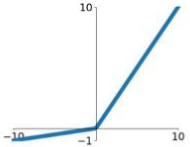
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

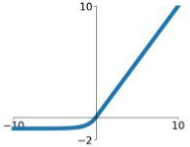


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

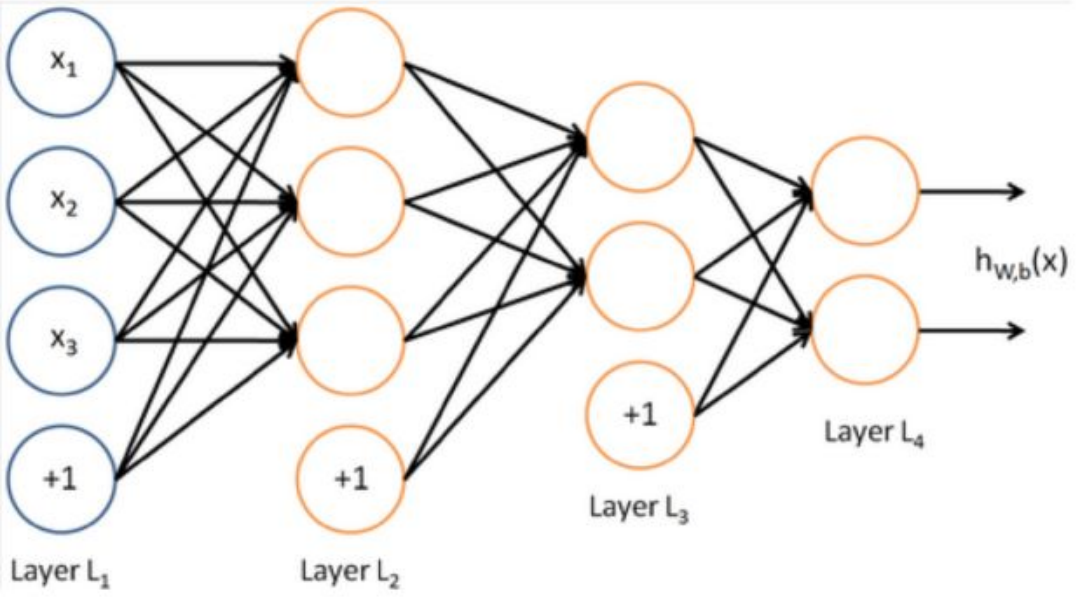
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Neural networks

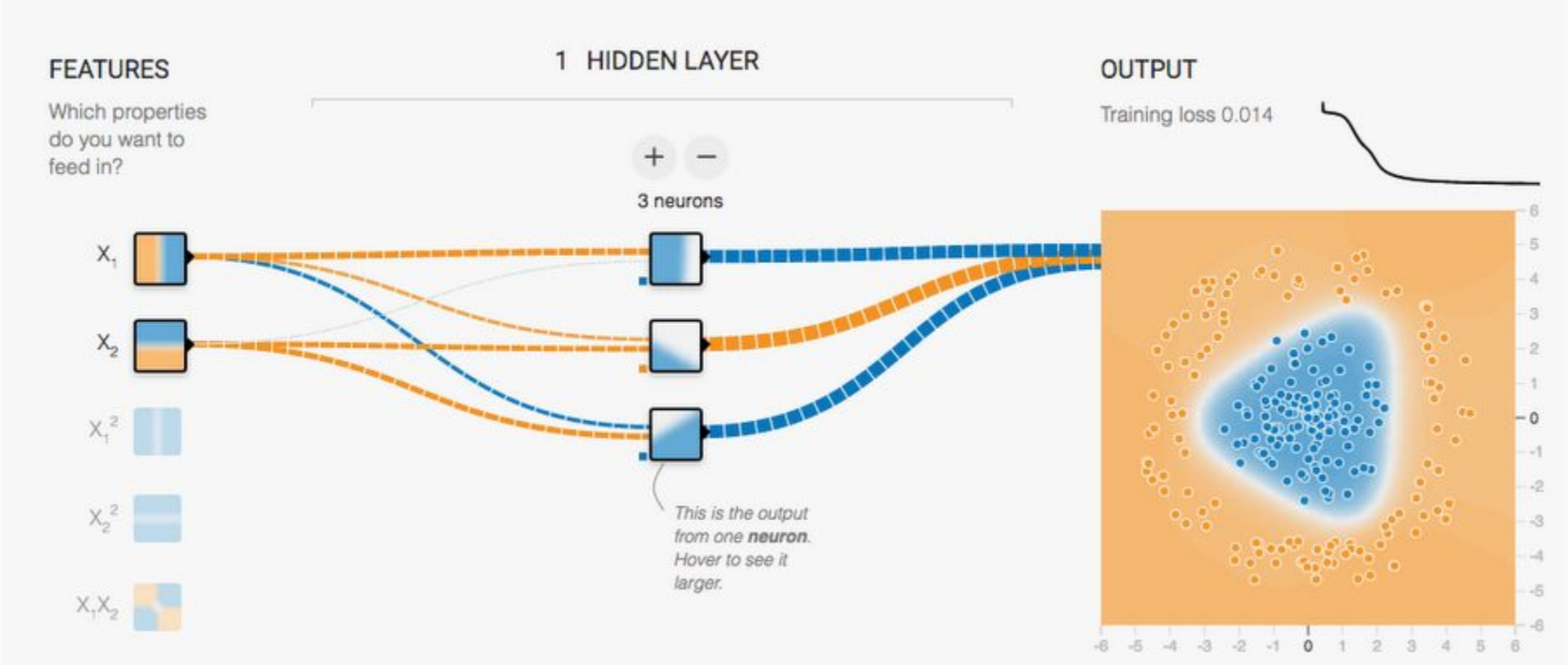
If a single unit is a logistic regression ($f : \mathbb{R}^n \to \mathbb{R}$), then a layer of a NN is a multivariate logistic regression ($f : \mathbb{R}^n \to \mathbb{R}^n$), and a multilayer NN is a concatenation of multivariate logistic regressions ($f \circ f \circ ... \circ f : \mathbb{R}^n \to \mathbb{R}^n$)



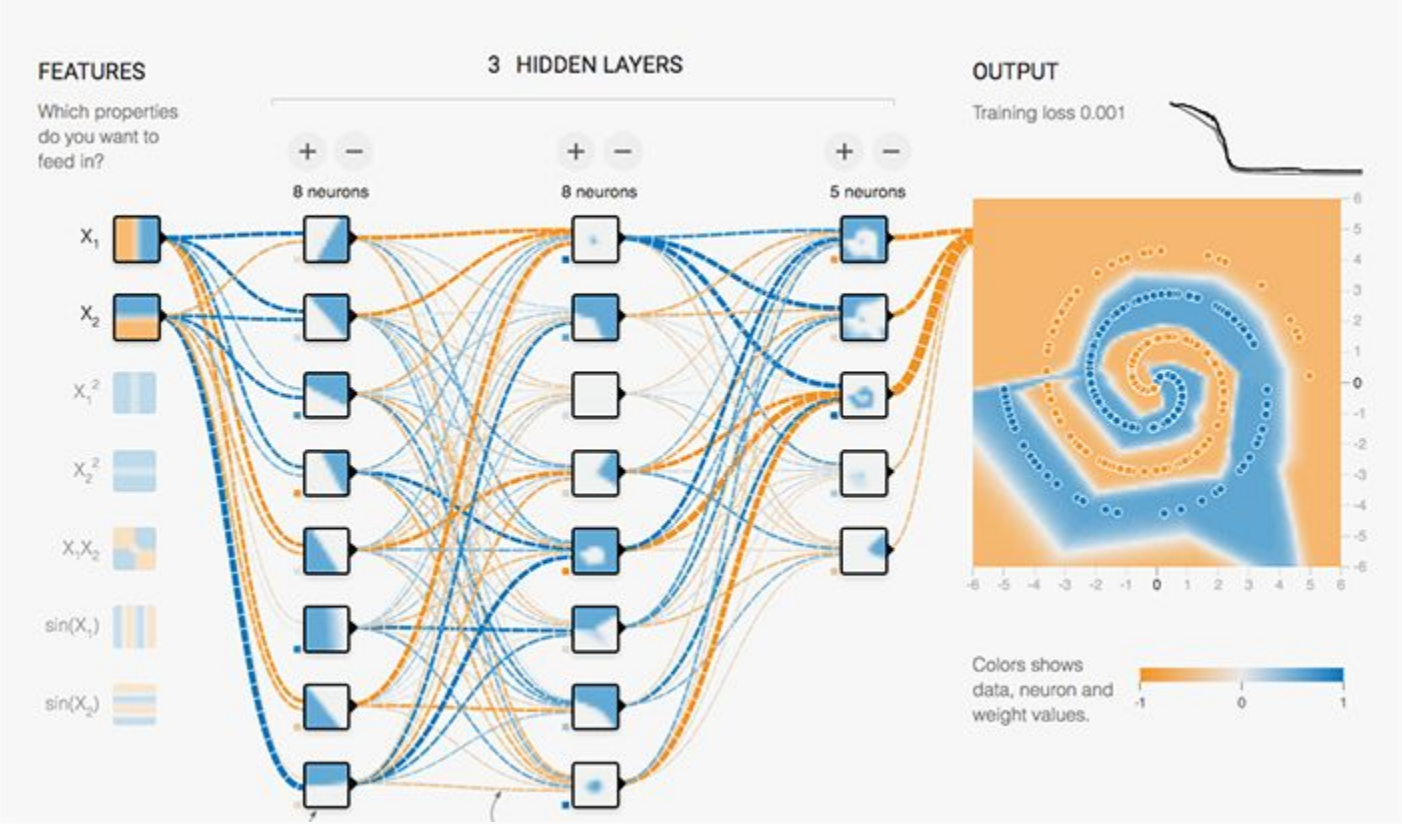
Neural networks

- Neural networks are concatenations of linear models with nonlinear transformation, and only with that they can theoretically represent any function (Google has an amazing tool to visualize and understand how neural networks do this)

Neural networks



Neural networks



→ Google Cloud tutorial by Kaz Sato: https://cloud.google.com/blog/products/gcp/understanding-neural-networks-with-tensorflow-playground

Named Entity Recognition (NER)

The problem: Very often (e.g. for many industrial applications) we would need to know what is the general classification (e.g. person, organization location, country...) of proper noun words within a piece of text. This is a typical NLP problem called Named Entity Recognition (NER)

The European Commission [ORG] said on Thursday it disagreed with German [MISC] advice.

Only France [LOC] and Britain [LOC] backed Fischler [PER] 's proposal .

“What we have to be extremely careful of is how other countries are going to take Germany 's lead”, Welsh National Farmers ' Union [ORG] (NFU [ORG]) chairman John Lloyd Jones [PER] said on BBC [ORG] radio .

Named Entity Recognition (NER)

Applications: Tracking mentions of a particular entity in documents; question answering (they often focus on named entities); finding relationships between entities; using the found entities and inputs to other more complex NLP tasks; others.

The European Commission [ORG] said on Thursday it disagreed with German [MISC] advice.

Only France [LOC] and Britain [LOC] backed Fischler [PER] 's proposal .

“What we have to be extremely careful of is how other countries are going to take Germany 's lead”, Welsh National Farmers ' Union [ORG] (NFU [ORG]) chairman John Lloyd Jones [PER] said on BBC [ORG] radio .

NER is hard

- Hard to know is something is an entity: Is “National Bank” the name of a bank? Or do they refer to “national bank” in the general sense of the word?

First National Bank Donates 2 Vans To Future School Of Fort Smith

POSTED 3:43 PM, JANUARY 11, 2019, BY SNEWS WEB STAFF

- Hard to know the boundaries of an entity: Is the name of the bank “National Bank” or “First National Bank”?
- Hard to know the class of a novel entity: What the heck is “Zig Ziglar”?

To find out more about Zig Ziglar and read features by other Creators Syndicate writers and

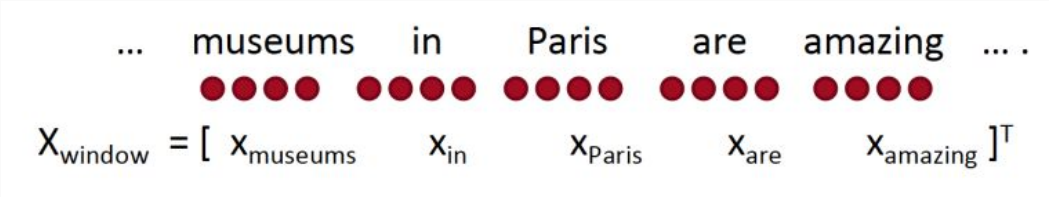
- Entity class can change in different contexts: Charles Schwab is a PER here, not ORG

where Larry Ellison and Charles Schwab can live discreetly amongst wooded estates. And

charles SCHWAB

NER: Applying multilayer NNs

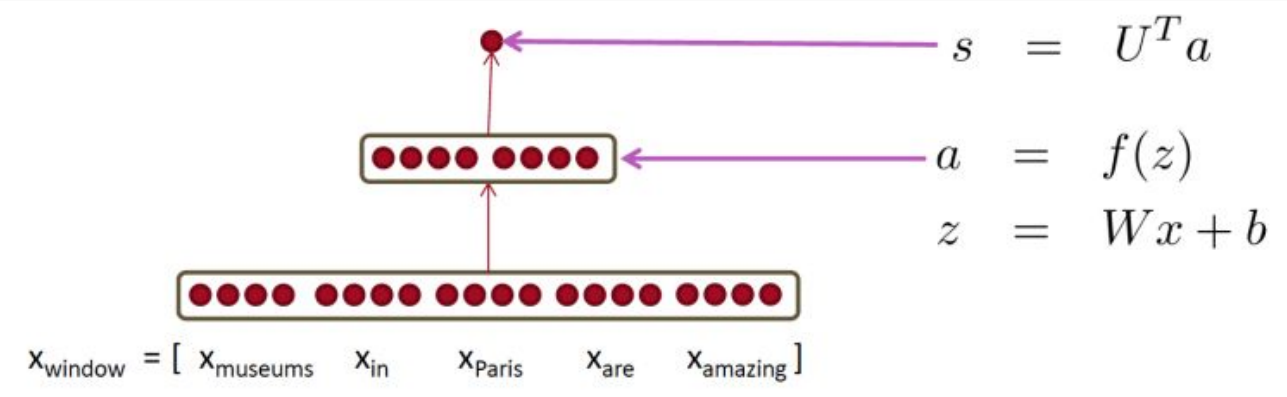
- The previous challenges likely stem from the fact that most of the information that determines whether a group of words is a Named Entity, and which is its entity class, resides in the context where that group of words is located. Most of this context is described in the neighbour words surrounding the named entity.
- **Idea:** Why don't we create a NN that reads a window of words around each word that we are evaluating?
- Example: "Not all museums in Paris are amazing"



- "Museums" tend to be inside cities (like Paris, capital of France), not people (like Paris Hilton, the actress)

NER: Applying multilayer NNs

- Input of the NN: $X_{window} = [\ x_{museums} \quad x_{in} \quad x_{Paris} \quad x_{are} \quad x_{amazing}\ ]$
- Output of the NN: whether the central word is [PER] (person), [LOC] (location) or no entity
- **Learning:** We will go over all words in a corpus, taking the embeddings of the neighbours as inputs to the NN (as we did in word2vec's CBOW), and the classification of the central word as output (different than in word2vec, where the output was the a word embedding).
- Architecture of the NN: 3 layers



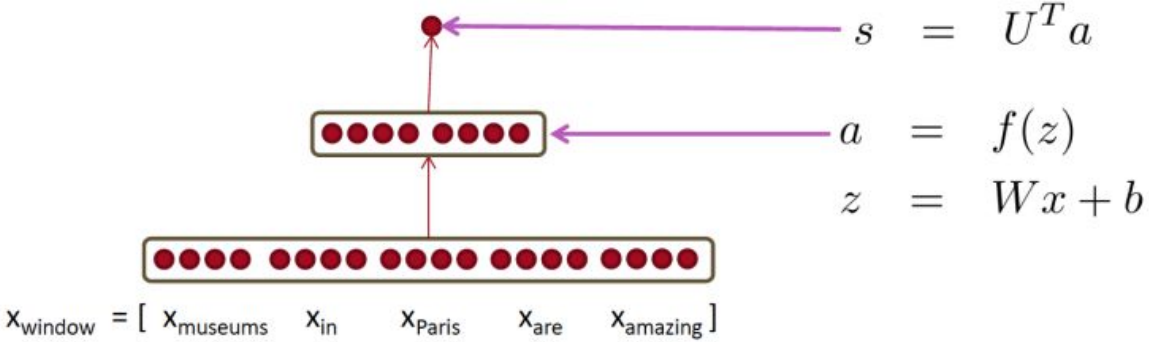
Method used by Collobert & Weston (2008, 2011). Won ICML 2018 test of time award

NER: Applying multilayer NNs

- The middle layer of the NN learns non-linear interactions between input words

$$s = U^T f(Wx + b)$$

$$x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$



- Example of interaction: only if “museums” is first vector, should it matter that “in” is in the second position

Applying multilayer NNs: Margin loss

- **Idea:** Use 2 types of input samples, one where the central word is you Named Entity (positive sample), and another one where the central word is not your Named Entity (negative sample)
  - $s_{POS} = U^T f(W x_{POS} + b)$ ← x_{POS} = museums in Paris are amazing
  - $s_{NEG} = U^T f(W x_{NEG} + b)$ ← x_{NEG} = Not all museums in Paris
- Then, take a positive and a negative sample, and minimize:
  - $J = \max(0, 1 + s_{NEG} - s_{POS})$ ← 0 only if $1 + s_{NEG} < s_{POS}$
  - Minimizing J will push the NN to make s_{POS} at least 1 point higher than s_{NEG}
- J is not everywhere differentiable (i.e. not differentiable in "$s_{NEG} - s_{POS} = 0$"), but it is continuous everywhere, so we can apply Stochastic Gradient Descent (SGD)
- This is similar to negative sampling in skip-gram

Applying multilayer NNs: Learning

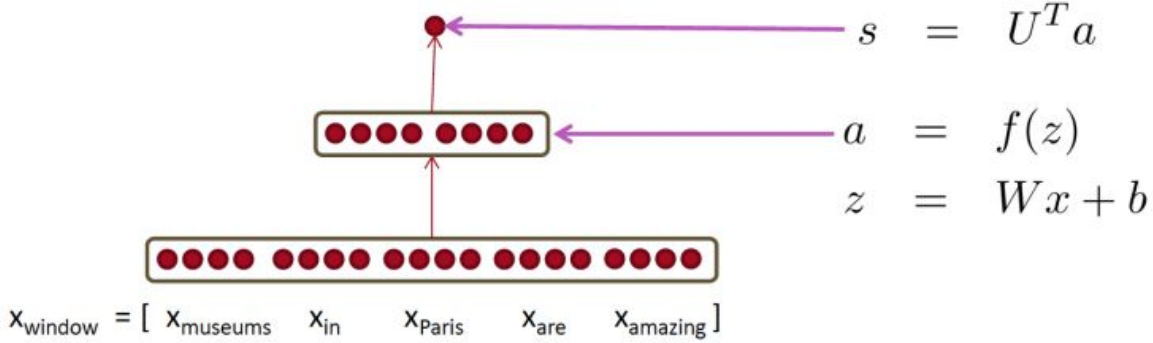
- Learning: SGD
  - 1) Feed forwards a number of inputs through the neural network → $s = U^T f(W x + b)$
  - 2) Calculate the total loss of the output produced by those inputs → $J = \dots x_{POS} \dots x_{NEG}$
  - 3) Calculate what "tiny changes" you should make to each parameters of the NN (U, W and b) for the loss J to be a "tiny bit" better
- How do we do step 3?: Differential calculus!
  - "tiny" change of variable 'a' → differential of 'a' → δa
  - How much 'b' changes if we change 'a' a "tiny" bit (δa) → derivative of 'b' with respect to 'a' → $\delta b / \delta a$
  - Chain rule of differential calculus = Given a concatenation of functions (like in a NN), we can calculate the derivatives of any variable given any other variable

Applying multilayer NNs: Learning

- How do we do step 3?: with the chain rule, we can calculate derivatives of any variable given any other variable

$$s = U^T f(Wx + b)$$

$$x \in \mathbb{R}^{20\times 1}, W \in \mathbb{R}^{8\times 20}, U \in \mathbb{R}^{8\times 1}$$



$\frac{\delta s}{\delta W}$ *How much 's' changes if we change 'W'*

Chain rule

$$\frac{\delta s}{\delta W} = \frac{\delta s}{\delta a}\frac{\delta a}{\delta z}\frac{\delta z}{\delta W}$$

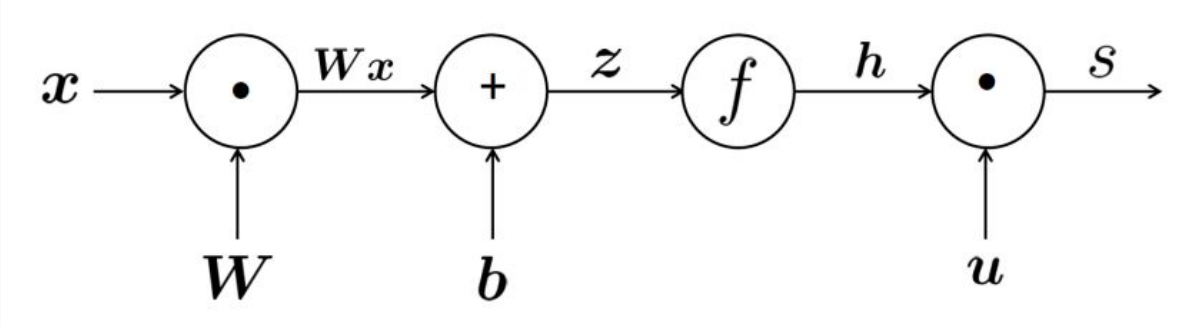
- And we can apply this not only to 's', but also to the loss function 'J' itself

Applying multilayer NNs: Backpropagation

- So, step 3 of SGD consist on:
  - 3.1) Calculate $\delta J/\delta U$, $\delta J/\delta W$ and $\delta J/\delta b$
  - 3.2) Modify U, W and b a tiny bit in the direction that maximises δJ. This is equivalent to: $W_{NEW} = W_{OLD} - \alpha \nabla_W J = W_{OLD} - \alpha \delta J/\delta W$
- But calculating the derivative of J with respect to every parameter (W and b) is very expensive:
  - U and W have as many rows as neurons in the next layer
  - W and b have as many columns as number of words in each window (e.g. 5) multiplied by the number of dimensions in the word embeddings (e.g. 200!). U has as many columns as neurons in the previous layer
  - And you have to do this many many times during learning!
- Solution: dynamics programming applied to NNs → Backpropagation!

Applying multilayer NNs: Backpropagation

- Idea: When calculating δJ/δU, δJ/δW and δJ/δb, many terms appear several times. Why don't we reuse the result every time we calculate any of these repeated terms?
- To best understand this, we represent the NN as a computation graph:
  - Graph inputs = NN inputs and NN parameters
  - Graph nodes = NN operations

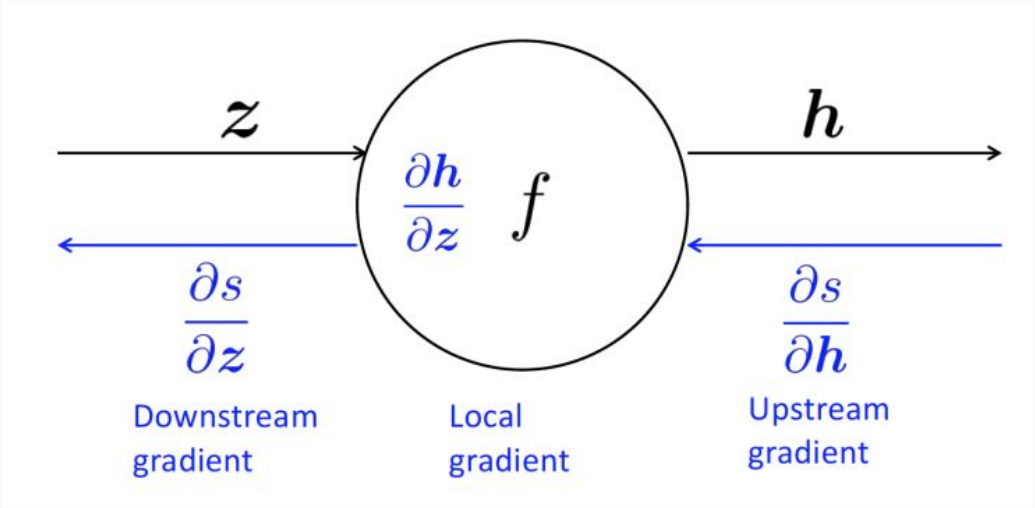


$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \quad \text{(input)}$$

Applying multilayer NNs: Backpropagation

- If you know how to apply backpropagation to 3 core simple graphs, to can apply them to all complex graphs:
  - Single node
  - Converging inputs
  - Diverging outputs

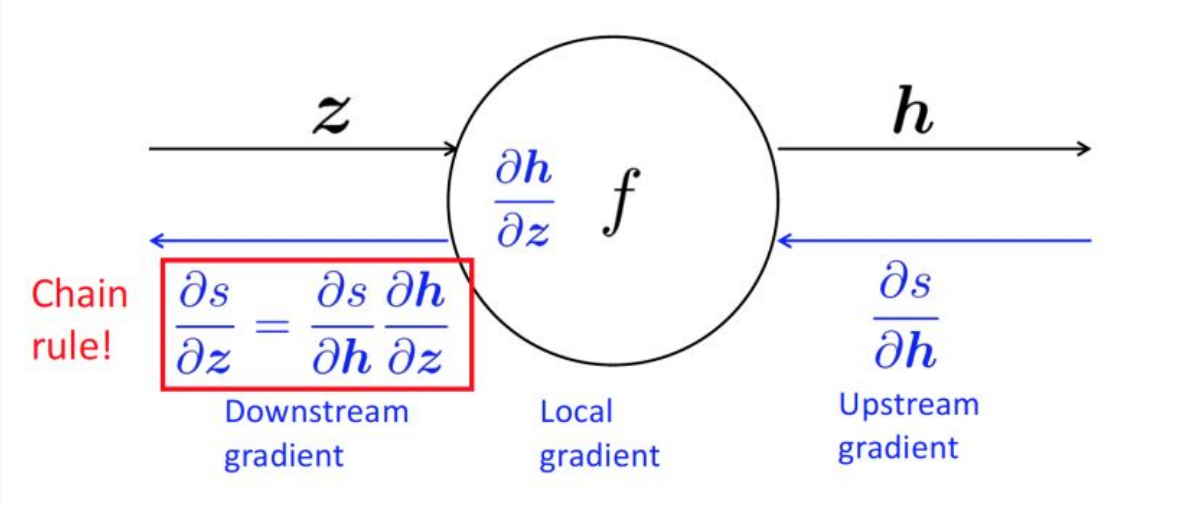
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Applying multilayer NNs: Backpropagation

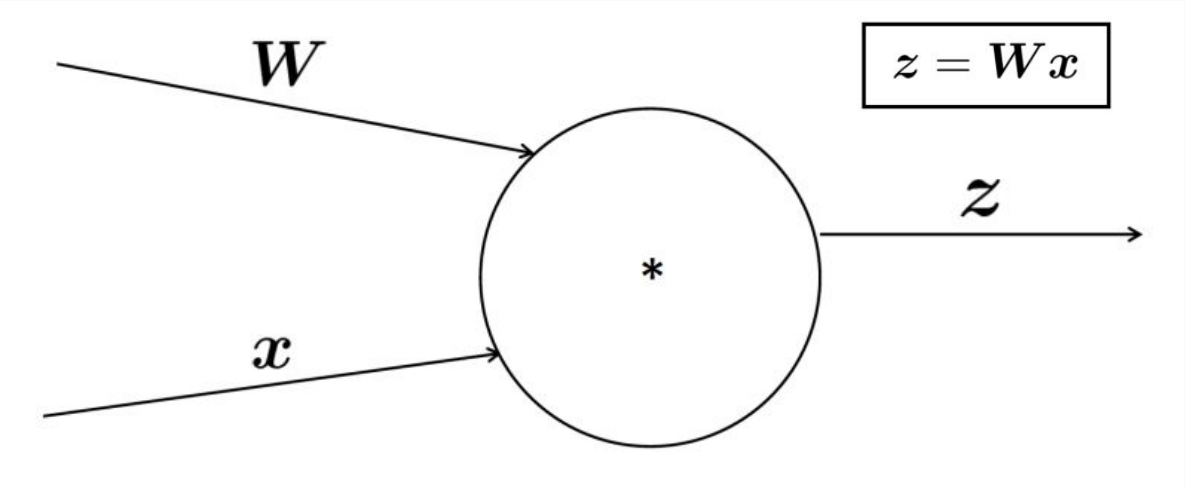
- If you know how to apply backpropagation to 3 core simple graphs, to can apply them to all complex graphs:
  - Single node
  - Converging inputs
  - Diverging outputs

$$\boldsymbol{h} = f(\boldsymbol{z})$$



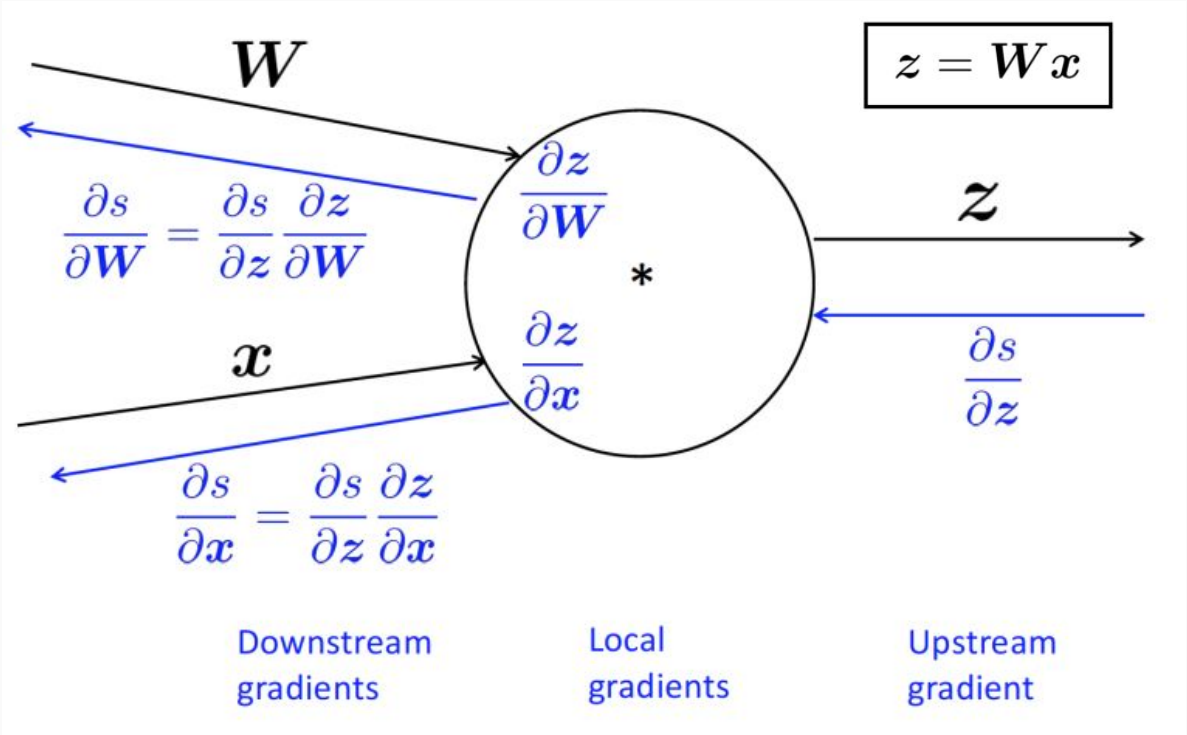
Applying multilayer NNs: Backpropagation

- If you know how to apply backpropagation to 3 core simple graphs, to can apply them to all complex graphs:
  - Single node
  - Converging inputs
  - Diverging outputs



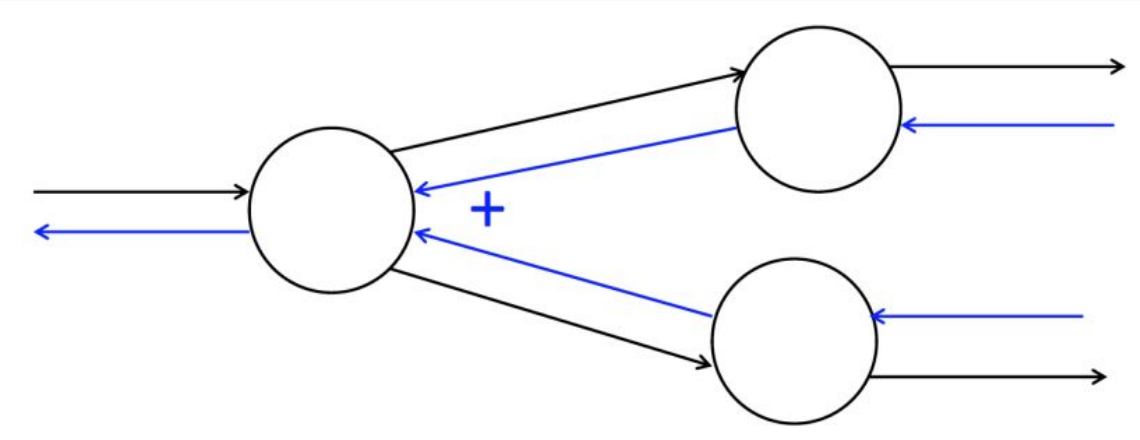
Applying multilayer NNs: Backpropagation

- If you know how to apply backpropagation to 3 core simple graphs, to can apply them to all complex graphs:
  - Single node
  - Converging inputs
  - Diverging outputs



Applying multilayer NNs: Backpropagation

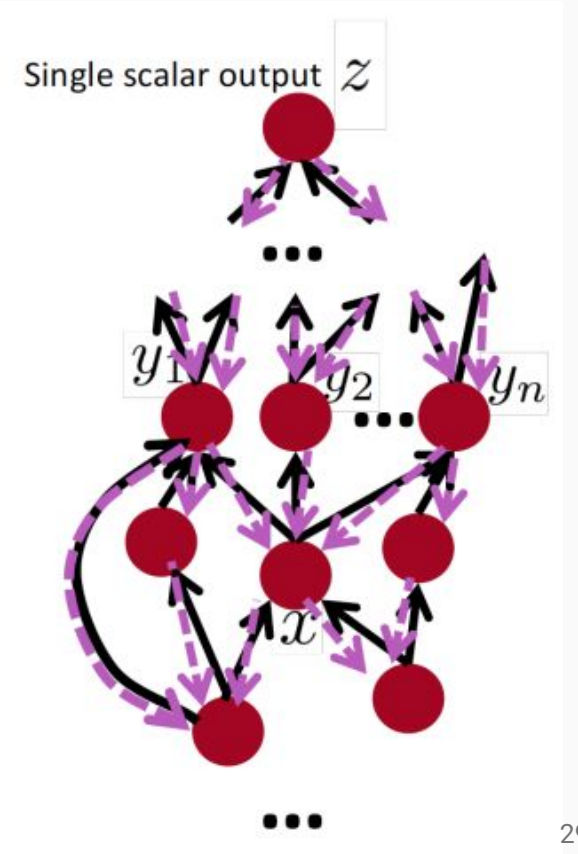
- If you know how to apply backpropagation to 3 core simple graphs, to can apply them to all complex graphs:
  - Single node
  - Converging inputs
  - Diverging outputs



Applying multilayer NNs: Backpropagation

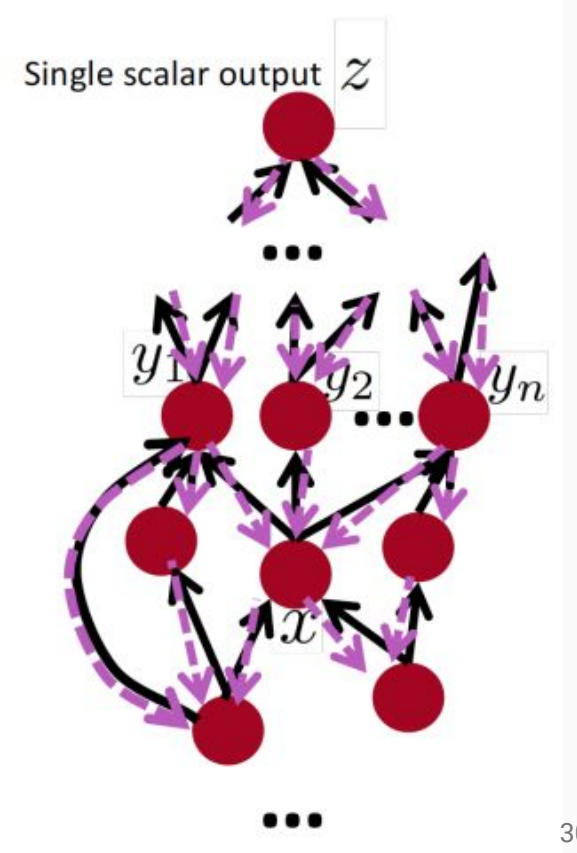
- Learning: SGD
  - 1) Feed forwards a number of inputs through the neural network → $s = U^T f(W x + b)$
  - 2) Calculate the total loss of the output produced by those inputs → $J = \dots x_{POS} \dots x_{NEG}$
  - 3) Calculate what "tiny changes" you should make to each parameters of the NN (U, W and b) for the loss J to be a "tiny bit" better
    - 3.1) Calculate $\delta J/\delta U$, $\delta J/\delta W$ and $\delta J/\delta b$ → Do this with backpropagation
    - 3.2) Modify U, W and b a tiny bit in the direction that maximises dJ. This is equivalent to:

$$W_{NEW} = W_{OLD} - \alpha \nabla_W J = W_{OLD} - \alpha \delta J/\delta W$$



Applying multilayer NNs: Backpropagation

- How backpropagation is implemented:
  - The gradient of the loss WRT any variable ‘a’ (δJ/δa) can be automatically calculated from how the loss relates to these variables (J = ...a…). You can represent all computational association between variables as a graph
  - Each node in the graph needs to know how to compute its output from its inputs during forward-propagation (y = f(x)), and how to compute the gradient of the loss WRT its inputs (δJ/δx) given the gradient of the loss WRT its outputs (δJ/δy) → δJ/δx = δJ/δy × δy/δx
  - This is how modern libraries work (pytorch, tensorflow)

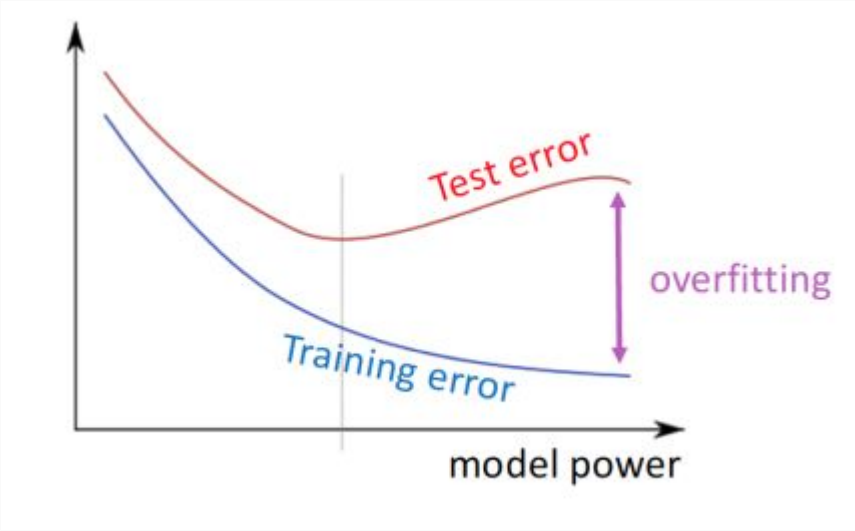


Applying multilayer NNs: Margin loss

- **Idea:** Use 2 types of input samples, one where the central word is you Named Entity (positive sample), and another one where the central word is not your Named Entity (negative sample)
  - $s_{POS} = U^T f(W x_{POS} + b)$ ← x_{POS} = museums in Paris are amazing
  - $s_{NEG} = U^T f(W x_{NEG} + b)$ ← x_{NEG} = Not all museums in Paris
- Then, take a positive and a negative sample, and minimize:
  - $J = \max(0, 1 + s_{NEG} - s_{POS})$ ← 0 only if $1 + s_{NEG} < s_{POS}$
  - Minimizing J will push the NN to make s_{POS} at least 1 point higher than s_{NEG}
- J is not everywhere differentiable (i.e. not differentiable in "$s_{NEG} - s_{POS} = 0$"), but it is continuous everywhere, so we can apply Stochastic Gradient Descent (SGD)
- This is similar to negative sampling in skip-gram

Applying multilayer NNs: Regularisation

- Problem: NNs tend to overfit because they have many parameters
- Idea 1: Add a penalisation to the loss function that encourages many parameters to take the value ‘0’ → L1 regularisation
- Idear 2: Add a penalisation to the loss function that encourages many parameters to take small values → L2 regularisation



$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

Applying multilayer NNs: Initialization

- Good basic initialization:
  - Initialize biases ‘b’ to 0
  - Initialize weights ‘W’ to random values from uniform distribution → Uniform(-r, +r)
  - Choose ‘r’ such that the values are not too large or too small
- Xavier initialization:
  - Choose initial values optimally for the variance of the information to stay stable across the NN
  - On initialization, the variance of the initial weights should be proportional to the size of the previous layer (fan-in, n_{in}) and to the size of the next layer (fan-out, n_{out})

$$\mathrm{Var}(W_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

Applying multilayer NNs: Optimizers

- Standard optimizer: SGD
  - This tends to work OK, but getting good results requires you to carefully fine-tune the learning rate
- Adaptive optimizers: They automatically fine-tune the learning rate, usually by monitoring the gradients during learning
  - Adagrad
  - RMSprop
  - Adam
  - Sparse Adam
  - ...

Course structure

- **Introduction:** What is NLP. Why it is hard. Why NNs work well ← Lecture 9 (NLP 1)
- **Word representation:** How to represent the meaning of individual words
  - Old technology: One-hot representations, synsets ← Lecture 9 (NLP 1)
  - Embeddings: First trick that boosted the performance of NNs in NLP ← Lecture 9 (NLP 1)
    - Word2vec: Single layer NN. CBOW and skip-gram ← Lecture 10 (NLP 2)
    - Co-occurrence matrices: Basic counts and SVD improvement ← Lecture 10 (NLP 2)
    - Glove: Combining word2vec and co-occurrence matrices idea ← Lecture 10 (NLP 2)
    - Evaluating performance of embeddings ← Lecture 10 (NLP 2)
- **Named Entity Recognition (NER):** How to find words of specific meaning within text
  - Multilayer NNs: Margin loss. Forward- and back-propagation ← Lecture 11 (NLP 3)
  - Better loss functions: margin loss, regularisation ← Lecture 11 (NLP 3)
  - Better initializations: uniform, xavier ← Lecture 11 (NLP 3)
  - Better optimizers: Adagrad, RMSprop, Adam... ← Lecture 11 (NLP 3)

Course structure

- **Language modelling:** How to represent the meaning of full pieces of text
  - Old technology: N-grams ← Lecture 12 (NLP 4)
  - Recursive NNs language models (RNNs) ← Lecture 12 (NLP 4)
  - Evaluating performance of language models ← Lecture 12 (NLP 4)
  - Vanishing gradients: Problem. Gradient clipping ← Lecture 13 (NLP 5)
  - Improved RNNs: LSTM, GRU ← Lecture 13 (NLP 5)
- **Machine translation:** How to translate text
  - Old technology: Georgetown–IBM experiment and ALPAC report ← Lecture 16 (NLP 6)
  - Seq2seq: Greedy decoding, encoder-decoder, beam search ← Lecture 16 (NLP 6)
  - Attention: Simple attention, transformers, reformers ← Lecture 16 (NLP 6)
  - Evaluating performance: BLEU ← Lecture 16 (NLP 6)

Literature

➢ Papers =

- “Automatic differentiation in machine learning: A survey”, Baydin et al., 2015. http://arxiv.org/abs/1502.05767
- “Natural language processing (almost) from scratch”, Collobert et al., 2011. http://www.jmlr.org/papers/volume12/collobert11a/collobert11a.pdf
- “Learning representations by backpropagating errors”, Rumelhart et al., 1986. http://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop old.pdf
- “Yes, you should understand backprop”, Karpathy, 2016. https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b