Linear models: Recap

Linear models:

- Perceptron

$$\text{score}(y, \boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$$

- Naïve Bay

$$\log P(y|\boldsymbol{x}; \boldsymbol{\theta}) = \log P(\boldsymbol{x}|y; \boldsymbol{\phi}) + \log P(y; \boldsymbol{\phi}) = B(\boldsymbol{x}) + \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y)$$

- Logistic Regression

$$\log P(y|\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta} \cdot \boldsymbol{f}(\boldsymbol{x}, y')$$

Features and weights in linear models: Recap

- Feature represen

$$\boldsymbol{f}(x, y=1) = [x; \underbrace{0; 0; \cdots; 0}_{(K-1)}]$$

$$\boldsymbol{f}(x, y=2) = [0; 0; \cdots; \underbrace{x; \cdots; x}_{2)\times V}]$$

- Weights: $\boldsymbol{\theta}$

$$\boldsymbol{\theta} = [\underbrace{\theta_1; \theta_2; \cdots; \theta_V}_{y=1}; \underbrace{\theta_1; \theta_2; \cdots; \theta_V}_{y=2}; \cdots; \underbrace{\theta_1; \theta_2; \cdots; \theta_V}_{y=K}]$$

Rearranging the features and weights

- Represent the features $\boldsymbol{x}$ as a *column* vector of length V, and represent the weights as a $\boldsymbol{\Theta}$ as K

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_V \end{bmatrix} \quad \boldsymbol{\Theta} = \begin{matrix} y = 1 \\ y = 2 \\ \cdots \\ y = K \end{matrix} \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & \cdots & \theta_{1,V} \\ \theta_{2,1} & \theta_{2,2} & \cdots & \theta_{2,V} \\ \cdots & \cdots & \cdots & \cdots \\ \theta_{K,1} & \theta_{K,2} & \cdots & \theta_{K,V} \end{bmatrix}$$

- What is $\boldsymbol{\Theta x}$?

Scores for each class

- Verify that $\psi_1, \psi_2, \ldots, \psi_K$ corres ach class

$$\boldsymbol{\Psi} = \boldsymbol{\Theta}\boldsymbol{x} =$$

Implementation in Pytorch

Digression: Matrix multiplication

- Matrix with m rows and n columns:

$$A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times o}, \quad p$$

where $C_{ij} =$

- Example:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} & & \\ 1 & 2 & 4 \end{bmatrix}$$

Digression: 3-D matrix multiplication

Tensor shape: (batch-size, sentence-length, embedding size)

SoftMax

- SoftMax, also kno .

$$\text{SoftMax}_i(\psi) = \frac{\exp \psi_i}{K}$$

for $i = 1, 2, \quad , K$

- Applying Sof
distribution

$$\text{SoftMax}(\Psi) = \begin{bmatrix} P(y=1) \\ P(y=2) \\ \dots \\ P(y=K) \end{bmatrix}$$

- Verify this is exactly logistic regression

Logistic regression as a neural network

$$\boldsymbol{y} = \text{SoftMax}(\boldsymbol{\Theta x})$$

$V = 5\, K = 3$

Going deep

- There is no reason w

$$\boldsymbol{z} = \sigma(\boldsymbol{\Theta}_1 \boldsymbol{x})$$
$$\boldsymbol{y} = \text{SoftMax}(\boldsymbol{\Theta}_2 \boldsymbol{z})$$

Going even deeper

- There is no reason w

$$\begin{aligned} \boldsymbol{z}_1 &= \sigma(\boldsymbol{\Theta}_1 \boldsymbol{x}) \\ \boldsymbol{z}_2 &= \sigma(\boldsymbol{\Theta}_2 \boldsymbol{z}_1) \\ \boldsymbol{y} &= \text{SoftMax}(\boldsymbol{\Theta}_3 \boldsymbol{z}_2) \end{aligned}$$

- But why?

Non-linear classification

Linear models like Logis high-dimensional vector space, and they are expressive enough and work well for many NLP problems, why do we need more complex non-linear models?

- There have been rapid advances in , a family of nonlinear me put through mult

- Deep learning facilitates the incorpora **ord embeddings**, which are dense vector r ords, that can be learned from massive amoun

  - It has evolved from early static embeddings (e.g., Word2vec, Glove) to recent dynamtic embeddings (ELMO, BERT, XLNet)
- Rapid advances in specialized hardware called graphic processing units (GPUs). Many deep learning models can be implemented efficiently on GPUs.

Feedforward Neural networks: an intuitive justification

- In image classification, instead of using the input (pixels) to predict the image type directly, you can imag
that you can predict the shapes of parts of an ima
hand, ear.
- In text process
say we want to classify movie reviews (or m)
into a label set of {Good, Bad, tead predicting these
labels directly, we first predict a set of comp
such as the story, acting, soundtrack, cinematography, etc.
from raw input (words in the text).

Face Recognition

Feedforward neural networks

Formally, this is what we do:

- **Use the text x to predict the featur** ecifically, train a logistic regression classifier to compute for each $k \in \{1, 2, \cdots, K_z$

- **Use the featu** Train a logistic regression cla unknown or hidden, so we will use the $P(z|$ eatures.

Caveat: it's easy to demonstrate what this is wha for image processing, but it's hard to show this is what's actually going on in language processing. Interpretability is a major issue in neural models for language processing.

The hidden layer: computing the composite features

- If we assume each z_k is binary, that is, $z_k \in \{0, 1\}$, then $P(z_k|\boldsymbol{x})$ can be modeled with binary logistic regression:

$$P(z_k = 1|\boldsymbol{x}; \boldsymbol{\Theta}^{(x\to z)}) = \sigma(\boldsymbol{\theta}_k \cdot \boldsymbol{x}) = (1 + \exp(-\boldsymbol{\theta}_k \cdot \boldsymbol{x}))^{-1}$$

- The weight matrix $\boldsymbol{\Theta}^{(x\to z)}$ can be constructed by stacking (not concatenating, as in linear regression) the weight vectors for each z_k:
$\boldsymbol{\Theta}^{(x\to z)} = \left[\boldsymbol{\theta}_1^{x\to z}, \boldsymbol{\theta}_2^{x\to z}, \ldots, \boldsymbol{\theta}_{K_z}^{x\to z}\right]$
- We assume an offset/bias term is included in $\boldsymbol{x}$ and its parameter is included in each $\boldsymbol{\theta}_k^{x\to z}$

Notations: $\boldsymbol{\Theta}^{(x\to z)} \in \mathbb{R}^{k_z \times V}$ is a real number matrix with a dimension of k_z rows and V columns

Activation functions

- Sigmoid: The rang).

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

Question: what's the value of the sigmoid fun $x = 0$?

- Tanh: The range of the tanh activation functi $-1, 1)$

Question: what's the value of the tanh fun $x = 0$?

- ReLU: The **rectified linear unit (ReLU)** is zero for negative inputs, and linear for positive inputs

$$ReLU(x) = max(x, 0) = \begin{cases} 0 & x < 0 \\ x & otherwise \end{cases}$$

Sigmoid and tanh are sometimes described as **squashing functions**.

Activation functions in Pytorch


```
from torch import nn
import torch

input = torch.ra  4)
sigmoid = nn.Si
output = sigmoi

tanh = nn.TanH()
output = tanh(input)

relu = nn.ReLU()
output = relu(input)
```

The output layer

- The output layer is computed by the multiclass logistic regression probability

$$P(y = j|\boldsymbol{z}; \boldsymbol{\Theta}^{(z\to y)}, \boldsymbol{b}) = \frac{e^{(\boldsymbol{\theta}_j^{z\to y} \cdot \boldsymbol{z} + b_j)}}{\sum_{j'} e^{(\boldsymbol{\theta}_{j'}^{(z\to y)} \cdot \boldsymbol{z} + b'_j)}}$$

- The weight matrix $\boldsymbol{\Theta} \in \mathbb{R}^{y \quad z}$ tructed by stacking weight vectors for each

$\boldsymbol{\Theta}^{(z\to y)} = \left[\boldsymbol{\theta}_1^{z\to y}, \boldsymbol{\theta}_2^{z\to y}, \ldots, \boldsymbol{\theta}_{K_y}^{z}\right.$

- The vector of probabilities over each possible value of y is denoted:
  $P(\boldsymbol{y}|\boldsymbol{z}; \boldsymbol{\Theta}^{(z\to y)}, \boldsymbol{b}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z\to y)}\boldsymbol{z} + \boldsymbol{b})$

The negative loglikelihood or cross-entropy loss

In a multi-class classifica probabilistic distribution over possible labels. It works well together with negative conditional likelihood (just like logistic regression)

$$= -\sum_{i=1}^{N} \log P(y^{(i)}$$

or **cross entropy lo**

$$\tilde{y}_j \triangleq Pr(y = j$$

$$-\mathcal{L} = -\sum_{i=1}^{N} \boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}}$$

where $\boldsymbol{e}_{y^{(i)}}$ is a **one-hot vector** of zeros with a value of one at the position $y^{(i)}$

Alternative loss functions

- There are alternatives to SoftMax and cross-entropy loss, just as there are alternatives in linear models.

- Pairing an affine transformation (remem h a margin loss:

$$\Psi(y; \boldsymbol{x}^{(i)}, \boldsymbol{\Theta}) = \boldsymbol{\theta}_y^{(z \to y)} \cdot \boldsymbol{z} + b_y$$

$$\ell_{\text{MARGIN}}(\boldsymbol{\Theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \max_{y \neq y^{(i)}} \left(1 + \Psi(y; \boldsymbol{x}^{(i)}, \boldsymbol{\Theta}) - \Psi(y^{(i)}; \boldsymbol{x}^{(i)}, \boldsymbol{\Theta})\right)_+$$

Training a feedforward network with the

https://eduassistpro.github.io/
Add WeChat edu_assist_pro

A feedforward network with a cross-entropy loss

The following sums up a feed-forward neural network with one hidden layer, complete with a cross-entropy loss th parameter estimation:

$$\boldsymbol{z} \leftarrow f(\boldsymbol{\Theta}^{x \to z} \boldsymbol{x})$$

$$\tilde{\boldsymbol{y}} \leftarrow \text{SoftMax}(\boldsymbol{\Theta}^{z \to y} \boldsymbol{z})$$

$$\ell^{(i)} \leftarrow -\boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}}$$

where f is an elementwise activation function (e.g., σ or ReLU), $\ell^{(i)}$ is the loss at instance i

Updating the parameters of a feedforward network

As usual, $\boldsymbol{\Theta}^{x \to z}$, $\boldsymbol{\Theta}^{z \to y}$, and $\boldsymbol{b}$ can be estimated by online gradient based optimization such as stochastic gradient descent:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \eta^{(t)} \nabla_{\boldsymbol{b}} \ell^{(i)}$$

$$\boldsymbol{\theta}_n^{x \to z} \leftarrow \boldsymbol{\theta}_n^{x \to z} - \eta^{(t)} \nabla_{\boldsymbol{\theta}_n^{x \to z}} \ell^{(i)}$$

where $\eta^{(t)}$ is the learning rate at iteration t, $\ell^{(i)}$ is the loss on instance (or minibatch) i , $\boldsymbol{\theta}_n^{x \to z}$ is the nth column of the matrix $\boldsymbol{\Theta}^{x \to z}$, and $\boldsymbol{\theta}_k^{z \to y}$ is the kth column of the matrix $\boldsymbol{\Theta}^{z \to y}$.

Compute the gradient of the cross-entropy loss on hidden layer weights $\Theta^{z\rightarrow}$

$$\nabla_{\boldsymbol{\theta}_k^{z\rightarrow y}} \ell^{(i)} = \left[\frac{\partial \ell^{(i)}}{\partial \theta_{k,1}^{z\rightarrow y}}, \frac{\partial \ell^{(i)}}{\partial \theta_{k,2}^{z\rightarrow y}}, \ldots\right]$$

$$\frac{\partial \ell^{(i)}}{\partial \theta_{k,j}^{z\rightarrow y}} = -\frac{\partial}{\partial \theta_{k,j}^{z\rightarrow y}} \left(\boldsymbol{\theta}_{y^{(i)}}^{(z\rightarrow y)} \cdot \boldsymbol{z} - \log \sum_{y\in\mathcal{Y}} \exp \boldsymbol{\theta}_y^{(z\rightarrow y)} \cdot \boldsymbol{z}\right)$$

$$= \left(P(y = j|\boldsymbol{z}; \boldsymbol{\Theta}^{(z\rightarrow y)}, \boldsymbol{b}) - \delta\left(j = y^{(i)}\right)\right) z_k$$

where $\delta\left(j = y^{(i)}\right)$ returns 1 if $j = y^{(i)}$ and 0 otherwise, z_k is the kth element in $\boldsymbol{z}$.

Applying the chain rule to compute the derivatives

We use the chain rule to break down the gradient of the loss on the hidden layer weights:

$$\frac{\partial \ell^{(i)}}{\partial \theta^{(z \to y)}} = \frac{\partial \ell^{(i)}}{\partial o} \frac{\partial o}{\partial \theta}$$

where

$$\ell^{(i)} = -\boldsymbol{e}_{y^{(i)}} \cdot \log \hat{\boldsymbol{y}} = -\log \left(\frac{}{\sum_k} \right) \log \sum_k e^{o_k}$$

$$o_j = \boldsymbol{\theta}_j^{(z \to y)} \boldsymbol{z}$$

Note: o_i is the logit that corresponds to the true label $y^{(i)}$

Derivative of the cross-entropy loss with respect to the logits

$$
\begin{aligned}
\frac{\partial \ell^{(i)}}{\partial o_j} &= \frac{\partial}{\partial o_j}\left(\log \sum_k e^{o_k} - o_{y^{(i)}}\right) \\
&= \frac{1}{\sum_k e^{o_k}} \frac{\partial}{\partial o_j} \sum_k e^{o_k} - \frac{\partial}{\partial o_j} o_{y^{(i)}} \\
&= \frac{e^{o_j}}{\sum_k e^{o_k}} - \frac{\partial}{\partial o_j} o_i \\
&= \tilde{y}_j - \delta(j = y^{(i)})
\end{aligned}
$$

Gradient on the hidden layer weights $\Theta^{z \to y}$

One more step to comput
the weights:

$$\frac{\partial o_j}{\partial \theta_{k,j}^{(z \to y)}} = \frac{\partial}{\partial \theta_{k,j}^{(z \to y)}} \boldsymbol{\theta}_j^{z \to y} \cdot \boldsymbol{z} = \frac{\partial}{\partial \theta^{(z \quad y)}} \theta_{k,j}^{z \to y} z_k = z_k$$

$$\frac{\partial \ell^{(i)}}{\partial \theta_{k,j}^{(z \to y)}} = \frac{\partial \ell^{(i)}}{\partial o_j} \frac{\partial o_j}{\partial \quad} = (\tilde{y} \quad \delta(j = y \quad)) z_k$$

Similarly, we can also compute the derivative w
hidden units:

$$\frac{\partial o_j}{\partial z_k} = \theta_{k,j}^{(z \to y)}$$

$$\frac{\partial \ell^{(i)}}{\partial z_k} = \frac{\partial \ell^{(i)}}{\partial o_j} \frac{\partial o_j}{\partial z_k} = (\tilde{y}_j - \delta(j = y^{(i)})) \theta_{k,j}^{(z \to y)}$$

This value will be useful for computing the gradient of the loss function from the inputs to the hidden layer.

Compute the gradient on the input weights $\Theta^{(x \to z)}$

Apply the old chain rule in differentiation:

$$\frac{\partial \ell^{(i)}}{\partial \theta_{jk}^{(x \to z)}} = \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial z_k}{\partial \theta_{jk}^{(x \to z)}}$$

$$= \frac{\partial \ell^{(i)}}{\partial z_k} f'\left(\theta_k^{(x \to z)} \cdot \boldsymbol{x}\right)$$

where $f'(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x})$ is the derivative f the activation function f, applied at the input $\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}$. Depending on what the actual activation is, the derivative will also be different.

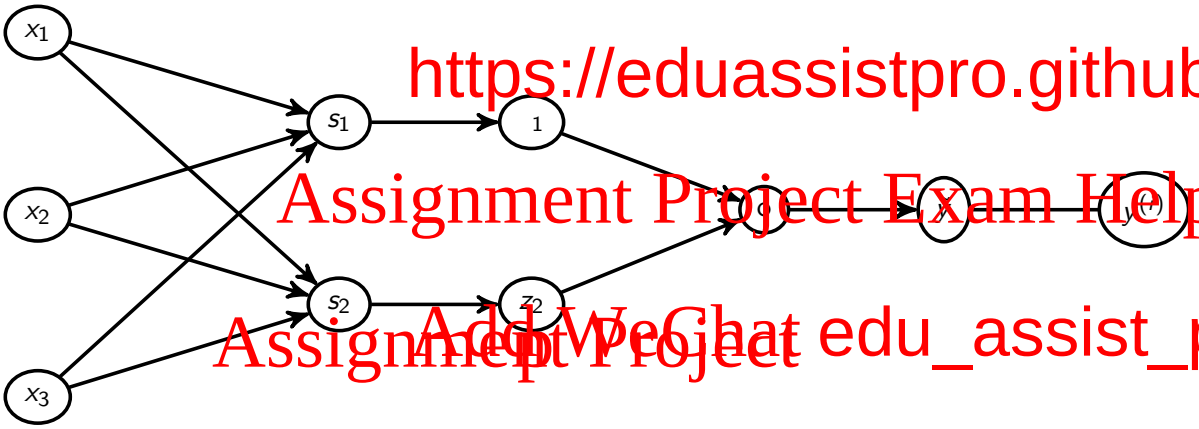
Derivative of the sigmoid activation function

$$\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{x \to z}} = \frac{\partial \ell^{(i)}}{\partial z_k} \sigma\left(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}\right)\left(1 - \sigma\left(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}\right)\right) x_n$$

- If the negative log-likelihood $\ell^{(i}$ [illegible] uch on z_k, then $\frac{\partial \ell^{(i)}}{\partial z_k} \approx 0$. In this case it doesn't mat[illegible] z_k is computed, and so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{x \to z}} \approx 0$

- If $x_n = 0$, then it does not matter how we set the weights $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{x \to z}}$, so $\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{x \to z}} = 0$

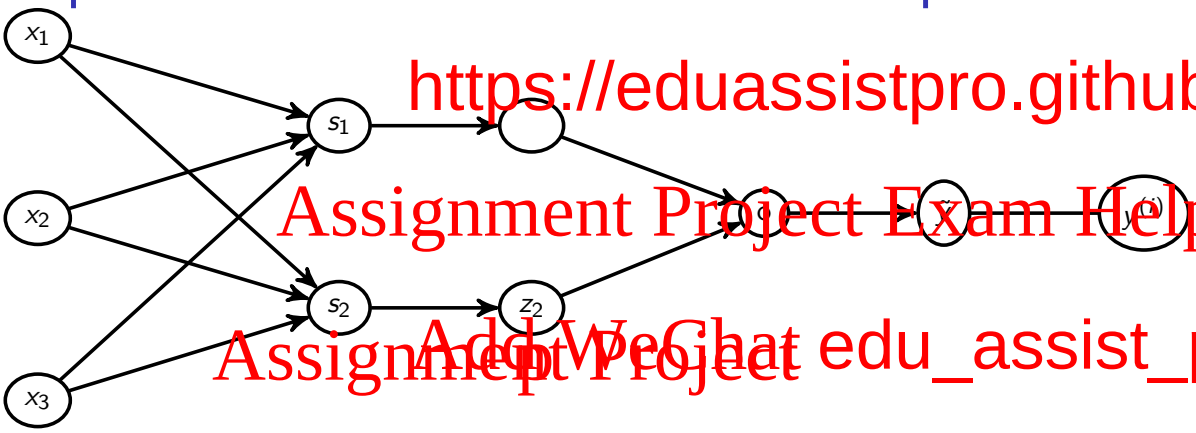
A simple neural network



$$\Theta^{(x\to s)} = \begin{array}{c} \\ s_1 \\ s_2 \end{array} \begin{array}{c} \begin{array}{ccc} x_1 & & x_3 \end{array} \\ \begin{bmatrix} \theta_{11}^{(x\to s)} & \theta_{12}^{(x\to s)} & \theta_{13}^{(x\to s)} \\ \theta_{21}^{(x\to s)} & \theta_{22}^{(x\to s)} & \theta_{23}^{(x\to s)} \end{bmatrix} \end{array}$$

$$\Theta^{(z\to o)} = \begin{array}{c} \\ o \end{array} \begin{array}{c} \begin{array}{cc} z_1 & z_2 \end{array} \\ \begin{bmatrix} \theta_{11}^{(z\to o)} & \theta_{12}^{(z\to o)} \end{bmatrix} \end{array}$$

A simple neural network: forward computation



$$s_1 = \theta_{11}^{(x \to s)} x_1 + \theta_{12}^{(x \to s)} x_2 + \theta_{13}^{(x \to s)} x_3$$

$$s_2 = \theta_{21}^{(x \to s)} x_1 + \theta_{22}^{(x \to s)} x_2 + \theta_{23}^{(x \to s)} x_3$$

$$z_1 = \sigma(s_1)$$

$$z_2 = \sigma(s_2)$$

$$o = \theta_{11}^{(z \to o)} z_1 + \theta_{12}^{(z \to o)} z_2$$

$$\tilde{y} = \sigma(o)$$

Note: When making predictions with the sigmoid function, choose the label 1 if $\tilde{y} > 0.5$. Otherwise choose 0

A simple neural network with cross-entropy sigmoid loss

- Cross-entropy sigmoid loss:

$$\mathcal{L}_{H(p,q)} = -\sum_i p_i \log q_i = -y \log(\tilde{y}) - (1-y)\log(1-\tilde{y})$$

- $y \in \{0, 1\}$ is the true label
- $\tilde{y}$ is the predicted value

- If the true label is 1, loss is $-\log(\tilde{y})$. Loss is bigger if $\tilde{y}$ is smaller
- If the true label is 0, loss is $-\log(1-\tilde{y})$. Loss is bigger if $\tilde{y}$ is bigger

A simple neural network: Compute the derivatives

$$\frac{\partial \mathcal{L}}{\partial o} = \tilde{y} - y$$

$$\frac{\partial o}{\partial z_1} = \theta_{11}^{(z \to o)} \quad \frac{\partial o}{\partial z_2} = \theta_{12}^{(z \to o)}$$

$$\frac{\partial o}{\partial \theta_{11}^{(z} \quad \frac{\partial o}{\partial}$$

$$\frac{\partial z_1}{\partial s_1} = z_1(1 - z_1) \quad \frac{\partial z_2}{\partial s_2} =$$

$$\frac{\partial s_1}{\partial \theta_{11}^{(x \to s)}} = x_1 \quad \frac{\partial s_1}{\partial \theta_{12}^{(x \to s)}} = x_2 \quad \frac{1}{\partial \theta_{13}^{(x \to s)}} = x_3$$

$$\frac{\partial s_2}{\partial \theta_{21}^{(x \to s)}} = x_1 \quad \frac{\partial s_2}{\partial \theta_{22}^{(x \to s)}} = x_2 \quad \frac{\partial s_2}{\partial \theta_{23}^{(x \to s)}} = x_3$$

Compute the derivatives on the weights $\Theta^{z \to o}$

Apply the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \theta_{12}^{(z \to o)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial \theta_{12}^{(z \to o)}} = (\tilde{y} \quad y) z_1$$

Compute derivatives with respect to the weights $\Theta^{x \to s}$

$$\frac{\partial \mathcal{L}}{\partial \theta_{11}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_1} \frac{\partial z_1}{\partial s_1} \frac{\partial s_1}{\partial \theta_{11}^{(x \to s)}} = (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_1$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{12}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_1} \frac{\partial z_1}{\partial s_1} \frac{\partial s_1}{\partial \theta_{12}^{(x \to s)}} = (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_2$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{13}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_1} \frac{\partial z_1}{\partial s_1} \frac{\partial s_1}{\partial \theta_{13}^{(x \to s)}} = (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_3$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{21}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_2} \frac{\partial z_2}{\partial s_2} \frac{\partial s_1}{\partial \theta_{21}^{(x \to s)}} = (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_1$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{22}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_2} \frac{\partial z_2}{\partial s_2} \frac{\partial s_1}{\partial \theta_{22}^{(x \to s)}} = (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_2$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{23}^{(x \to s)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\partial z_2} \frac{\partial z_2}{\partial s_2} \frac{\partial s_1}{\partial \theta_{23}^{(x \to s)}} = (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_3$$

Vectorizing forward computation

$$\boldsymbol{s} = \boldsymbol{\Theta}^{(x \to s)} \boldsymbol{x} = \begin{bmatrix} \theta_{11}^{(x \to s)} & \theta_{12}^{(x \to s)} & \theta_{13}^{(x \to s)} \\ \theta_{21}^{(x \to s)} & \theta_{22}^{(x \to s)} & \theta_{(x \to s)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\theta_{11}^{(x \to s)} x_1 + \theta_{12}^{(x \to s)} \quad 2 \quad 13 \quad 3 \quad s) x_3 = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$$

$$\boldsymbol{z} = \sigma\left(\begin{bmatrix} s_1 \\ s_2 \end{bmatrix}\right) = \begin{bmatrix} \sigma(s_1) \\ \sigma(s_2) \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

$$\boldsymbol{o} = \boldsymbol{\Theta}^{(z \to o)} \boldsymbol{z} = \begin{bmatrix} \theta_{11}^{(z \to o)} & \theta_{12}^{(z \to o)} \end{bmatrix} \times \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} \theta_{11}^{(z \to o)} z_1 + \theta_{12}^{(z \to o)} z_2 \end{bmatrix}$$

$$\tilde{y} = \sigma(\boldsymbol{o})$$

Note: Summing over inputs in forward computation

Backpropagation

- When applying the chain rule, local derivatives are frequently reused. For example, $\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}}$ is used to compute the gradient on both $\boldsymbol{\Theta}^{z \to o}$ and $\boldsymbol{\Theta}^{x \to s}$. It would be usefu **he them**.
- We can only compute the derivatives of the par we have all the ne rule. So carefuri s e of the cached derivatives.
- This combination of *sequencing* d *differentiation* is called **backpropagation**.
- In order to make this process manageable, backpropagation is typically done in vectorial form (tensors)

Elements in a neural network

- Variables includes input $\boldsymbol{x}$, hidden units $\boldsymbol{z}$, outputs $\boldsymbol{y}$, and the loss function.

  - Inputs are not computed from other nodes i

example, inputs to a feed-forward neural n

feature vector extracted from a training (o
  - Backpr

espect to
all variab
eters.
- Parameters include weights and bias. T

parents, and are initialized and then upd
descent
- Loss is not used to compute any other nodes in the graph, and is usually computed with the predicted label $\hat{y}$ and the true label $y(i)$.

Backpropagation: caching "error signals"

Let $\boldsymbol{D}_o$ represent the gradient of the loss function with respect to $\boldsymbol{o}$ and $\boldsymbol{D}_s$ be the gradient of the loss function with respect to $\boldsymbol{s}$. Assuming a cross-entropy loss and a sigmoid (whic genralized to Softmax) function, the error signal a $\boldsymbol{o}$ is:

$\boldsymbol{D_o} = [\tilde{y} - y^{(i)}]$

This error signal can t respect to $\boldsymbol{\Theta}^{(z\rightarrow}$

$$\nabla_{\boldsymbol{\Theta}^{(z\rightarrow o)}} = \boldsymbol{D_o z} = \begin{bmatrix}\tilde{y} - y\end{bmatrix} \begin{bmatrix} z_1 & z_2 \end{bmatrix}$$
$$= \begin{bmatrix}(\tilde{y} - y)z_1 & (\tilde{y} - y)z_2\end{bmatrix}$$

Backpropagation: Computing error signals

Using $\boldsymbol{D_o}$, we can also compute the error signals at the hidden layer:

$$\boldsymbol{D_s} = (\boldsymbol{\Theta}^{(z\to o)}$$

$$= \left(\left[\frac{\partial o}{\partial z_1} \quad \frac{\partial o}{\partial z_2} \right] \quad \overline{\frac{\partial o}{\partial s_1}} \quad \overline{\quad}$$

$$= \left(\left[\theta_{11}^{(z\to o)} \quad \theta_{12}^{(z\to o)} \right] \times [\tilde{y} - y] \right] \quad z_2(1 - z_2)]$$

$$= \left[(\tilde{y} - y)\theta_{11}^{(z\to o)} z_1(1 - z_1) \quad (\tilde{y} - y)\theta_{12}^{(z\to o)} z_2(1 - z_2) \right]$$

Backpropagation: caching "error signals"

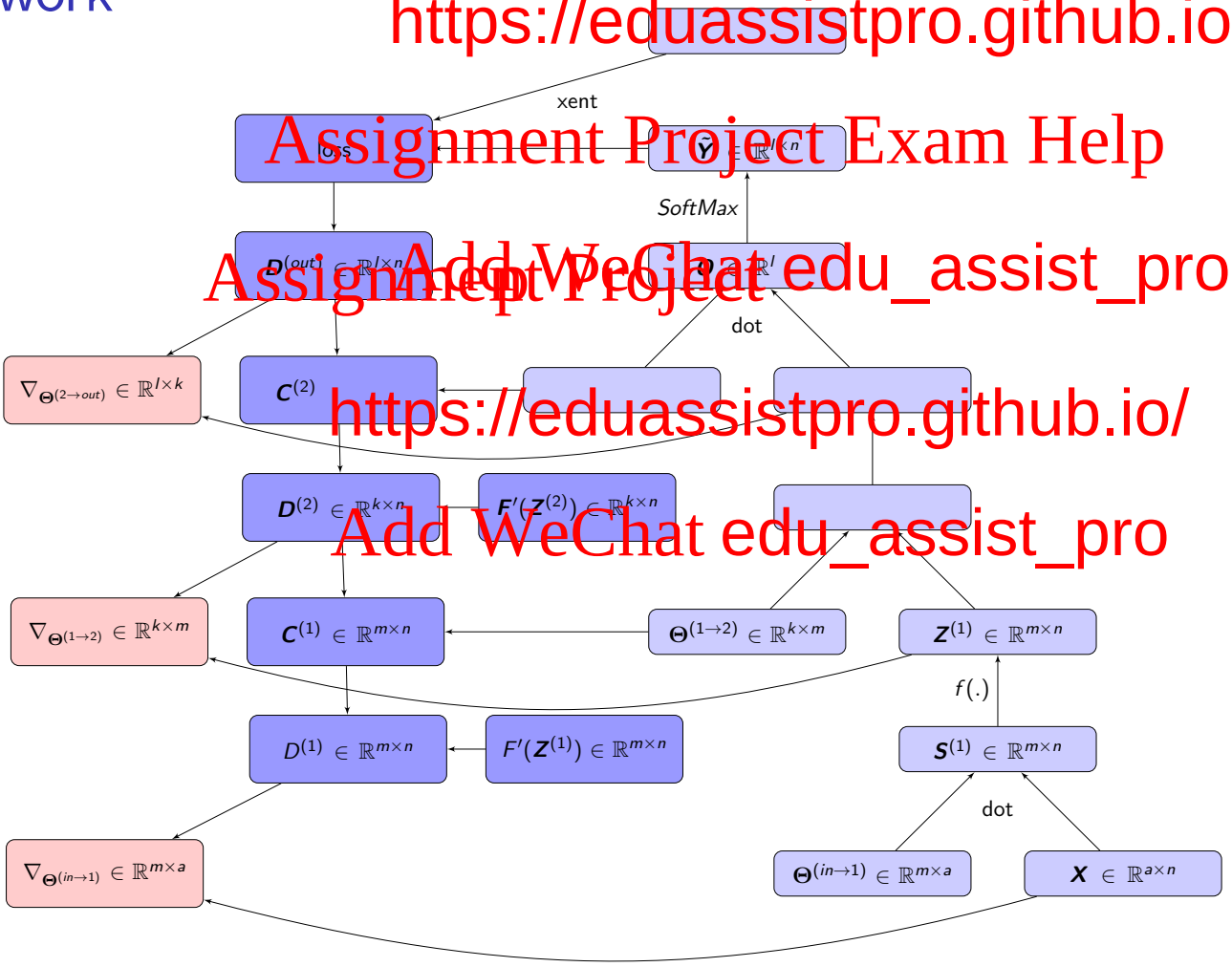
Using error signals $\boldsymbol{D_s}$, we can now compute th $\boldsymbol{\Theta}^{(x \to s)}$:

$$\nabla_{\boldsymbol{\Theta}^{(x \to s)}} = \boldsymbol{D_s}^\top \boldsymbol{X}$$

$$= \begin{bmatrix} (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1 \\ (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2) \end{bmatrix}$$

$$= \begin{bmatrix} (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_1 & (\tilde{y} - y)\theta_{11}^{(z \to o} & & - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_3 \\ (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_1 & (\tilde{y} - y)\theta_{12}^{(z \to o} & {}_2 \quad {}_2 \quad {}_2 & - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_3 \end{bmatrix}$$

Computation graph of a three-layer feedforward neural network



Forward computation in matrix form

$$\boldsymbol{S}^{(1)} = \boldsymbol{\Theta}^{in \to 1} \boldsymbol{X}$$

$$\boldsymbol{Z}^{(1)} = f_1(\boldsymbol{S}^{(1)})$$

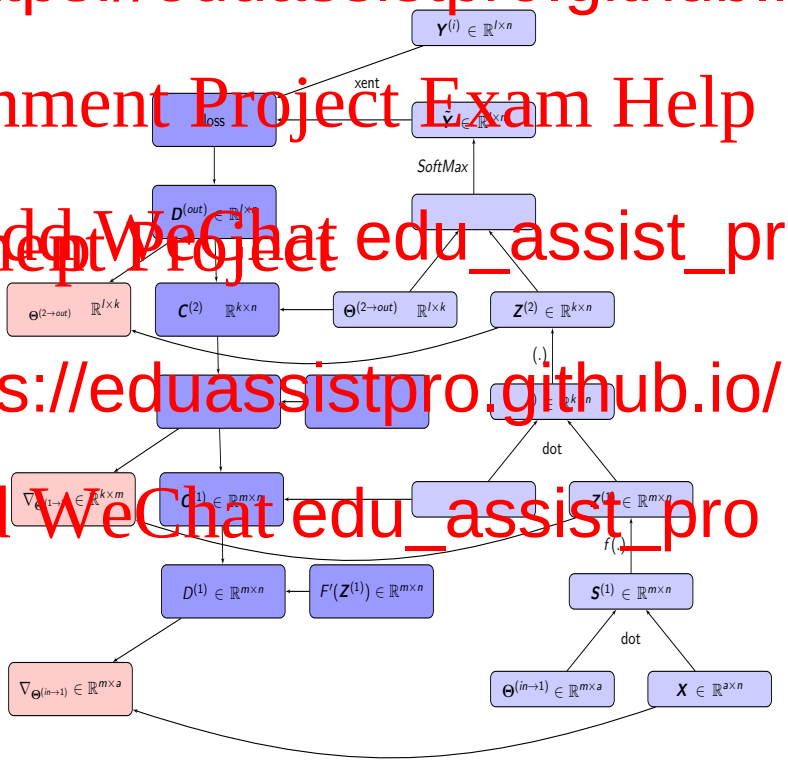
$$\boldsymbol{S}^{(2)} = \boldsymbol{\Theta}^{1 \to 2} \boldsymbol{Z}^{(1)}$$

$$\boldsymbol{Z}^{(2)} = f_2(\boldsymbol{S}^{(2)})$$

$$\boldsymbol{O} = \boldsymbol{\Theta}^{2 \to o} \boldsymbol{Z}^{(2)}$$

$$\tilde{\boldsymbol{y}}_{[j]} = \text{SoftMax}(\boldsymbol{O}^{\top}_{[j]})$$

Note SoftMax applies to a vector.



Matrix multiplication in forward computation sums over inputs, hidden units. We assume the input and hidden units in batches

Backpropagation: Computing error signals

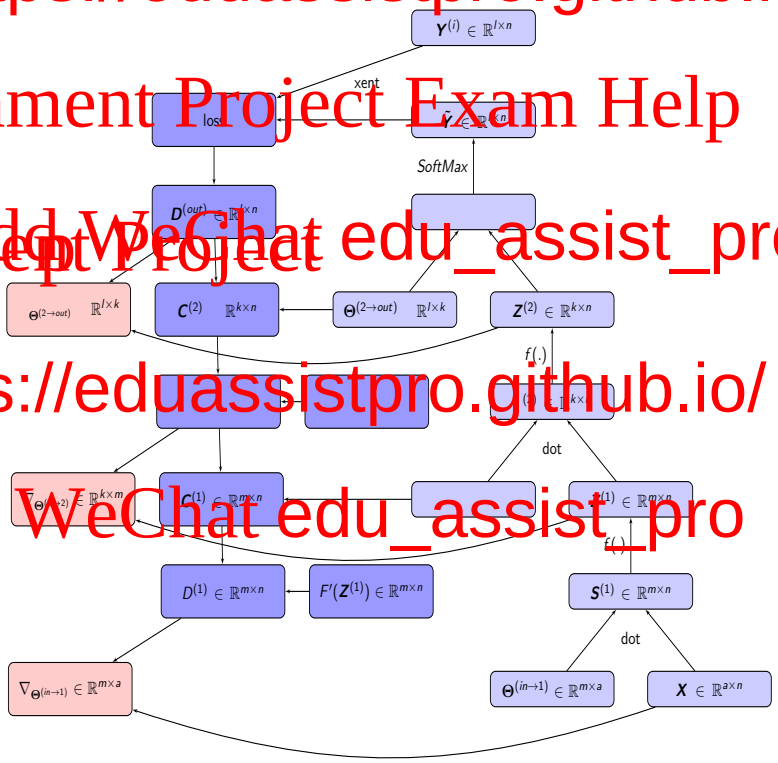
$$\boldsymbol{D}^{out} = \tilde{\boldsymbol{Y}} - \boldsymbol{Y}^{(i)}$$

$$\boldsymbol{C}^{(2)} = (\boldsymbol{\Theta}^{2\to o})^\top \boldsymbol{D}^{out}$$

$$\boldsymbol{D}^{(2)} = \boldsymbol{F}'(\boldsymbol{Z}^{(2)}) \odot \boldsymbol{C}^{(2)}$$

$$\boldsymbol{C}^{(1)} = (\boldsymbol{\Theta}^{1\to 2})^\top \boldsymbol{D}^{(}$$

$$\boldsymbol{D}^{(1)} = \boldsymbol{F}'(\boldsymbol{Z}^{(1)}) \odot$$



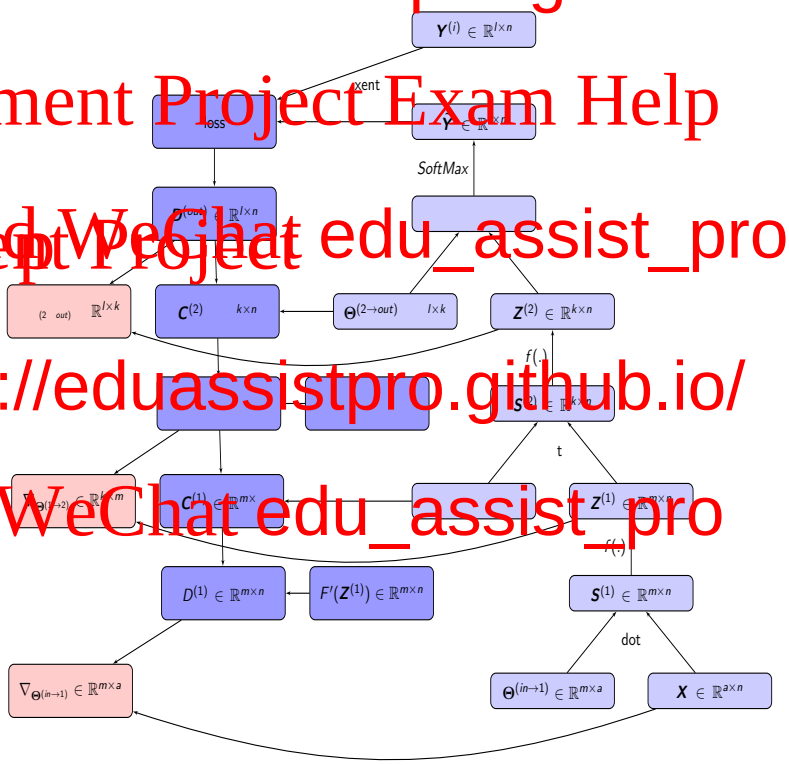
Matrix multiplication in error signal computation involves summing over outputs and hidden units.

Backpropagation: Compute the gradient with error signals

$$\nabla \boldsymbol{\Theta}^{2 \to out} = \boldsymbol{D}^{(out)} (\boldsymbol{Z}^{(2)})^\top$$

$$\nabla \boldsymbol{\Theta}^{1 \to 2} = \boldsymbol{D}^{(2)} (\boldsymbol{Z}^{(1)})^\top$$

$$\nabla \boldsymbol{\Theta}^{in \to 1} = \boldsymbol{D}^{(1)} (\boldsymbol{X})^\top$$



Matrix multiplication when computing gradient sums over instances in mini-batch.

Notes in backpropagation

- Where does caching take place?
- Why is sequen
- What computation patterns can you ob

Pay attention to what you sum over

- In forward computation, you sum over all inp
  vector) for each output
- In backward c ll
  outputs for ea
- When you com ver
  the gradient for each data point (and aver
- Make sure you line up the columns of the firs
  rows of the second matrix. That's what you sum over.

Automatic differentiation software

- Backpropagation is mechanical. There is no reason for everyone to repeat the same work
- Currently many alternatives to Theano exist, e.g., Tor
- Using these libraries, the computation, and the gradient can be co automatically. These libraries have ac and development in deep learning consider
- Libraries that support dynamic computation graphs are better suited for many NLP problems.

Assignment Project

Bells an

Tricks in training neural networks

There are various tricks that people use when traini
networks:

- Regularizati
- Dropout: Adj
- Optimization methods: Adjusting the l
- Initialization: Using particular forms o i

Regularization

Neural networks can be regularized in a similar way as linear models. Neural networks can also with **Frobenius norm**, which is a trivial extension to L2 norm for matrices. In fact, in ma it is just referred to as L2 regularization.

$$\mathcal{L} = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z \to y} \|\mathbf{\Theta}^{(z \to y} \qquad ^{(x \to z)}\|_F^2$$

where $\|\mathbf{\Theta}\|_F^2 = \sum_{i,j} \theta_{i,j}^2$ is the squred **Frobenius norm**, which generalizes the L_2 norm to matrices. The bias parameters b are not regularized, as they do not contribute to the classifier to the inputs.

L2 regularization

- Compute the grad

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial \theta} + \lambda\theta$$

- Update the we

$$\theta = \theta - \eta \left(\sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial \theta} \right)$$

- "Weigh decay factor": λ is a tunable hyper parameter that pulls a weight back when it has become too big
- Question: Does it matter which layer θ is from when computing the regularization term?

L1 regularization

- L1 regularizatio

$$\mathcal{L} = \sum_{i=1}^{N} \ell^{(i)} + \lambda_{z \to y} \|\mathbf{\Theta}^{(z \to y)} \quad {}^{(x \to z)}\|_1$$

- Compute the g

$$\frac{\partial}{\partial \theta} \sum_{i=1} \frac{\partial}{\partial \theta}$$

- update the weights

$$\theta = \theta - \eta \left(\sum_{i=1}^{N} \frac{\partial \ell^{(i)}}{\partial \theta} + \lambda \, sign(\theta) \right)$$

Comparison of L1 and L2

- In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w.
- When a particular weight has a large absolute $|\theta|$, L1 regularizati
  regularizati small, L1
  regularizati
  regularization.
- The net result is that L1 regularization ten
  the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero. So L1 regularization effectively does *feature selection*.

Dropout

- Randomly drops a certain percentage of the n
  over-relianc **feature**
  co-adaptaten
  working toge goal
  is to avoid overfitting.

Dropout

- Dropout can be ach

$$\boldsymbol{z}^{(1)} = g^1(\boldsymbol{\Theta}^{(1)}\boldsymbol{x} + \boldsymbol{b}^1)$$

$$\boldsymbol{m}^1 \sim Bernouli($$

$$\tilde{\boldsymbol{z}}^{(1)} = \boldsymbol{m}^1 \quad \boldsymbol{z}^{(1}$$

$$\boldsymbol{m}^2 \sim Berno^{2}$$

$$\tilde{\boldsymbol{z}}^{(2)} = \boldsymbol{m}^2 \odot$$

$$\boldsymbol{y} = \boldsymbol{\Theta}^{(3)}\tilde{\boldsymbol{z}}^{(2)}$$

where $\boldsymbol{m}^1$ and $\boldsymbol{m}^2$ are mask vectors. The values of the elements in these vectors are either 1 or 0, drawn from a Bernouli distribution with parameter r (usually $r = 0.5$)

Optimization methods

- Moment
- Adgrad
- Root Mean Sq
- Adam

Momentum

At each timestep t, compute $\nabla_{\Theta} J$ and $\Delta\Theta$, and then compute the momentum as follows:

$$\Theta = \Theta - \quad_t$$

The momentum term increases for dimensio
point in the same directions and reduces updates for dimensions whose gradient change directions.

Adgrad

Weight and bias update for Adgrad at each time step t:

e.g., $\epsilon = 10^{-8}$

Root Mean Square Prop (RMSProp)

Weight update for RMSprop at each time step t:

$$\Delta w_t = -\frac{\eta}{\sqrt{s_t + \epsilon}} \frac{\partial L}{\partial w_t}$$

$$s_t = \beta s_{t-1} + (1-\beta)\left(\frac{\partial L}{\partial w_t}\right)^2$$

e.g. $\beta = 0.9, \eta = 0.001, \epsilon = 10^{-8}$

Adaptive Moment Estimation (Adam)

Weight update at time st

$$\boldsymbol{V}_{\nabla_\Theta} = \beta_1 \boldsymbol{V}_{\nabla_\Theta} + (1$$

$$\boldsymbol{S}_{\nabla_\Theta} = \beta_2 \boldsymbol{S}_{\nabla_\Theta} + (1 - \beta_2) \nabla_\Theta \nabla_\Theta$$

$$\boldsymbol{S}^{corrected}_{\nabla_\Theta} = \frac{\boldsymbol{S}_{\nabla_\Theta}}{\beta_2^t}$$

$$\boldsymbol{\Theta} = \boldsymbol{\Theta} - \eta \frac{\boldsymbol{V}^{corrected}_{\nabla_\Theta}}{\sqrt{\boldsymbol{S}^{corrected}_{\nabla_\Theta}} + \epsilon}$$

Adam combines Momentum and RMSProp

Define a neural net

```
from torch i
class Net(nn.
    '''subclass from nn.Module is important to insp
    def __init__(self, in_dim=25, out_dim=3, batch_
        super(Net, self).__i
        self.in_dim = in_dim
        self
        self                                  _dim, self.o
        self.softmax = nn.                    im=1) #softmax o

    def forward(self, input_
        logit = self.linear(input_matrix)
        return logit #return raw score, not normali

    def xtropy_loss(self, input_matrix, target_labe
        loss = nn.CrossEntropyLoss()
        logits = self.forward(input_matrix)
        return loss(logits,target_label_vec)
```

Use optimizers in Pytorch

```
import torch.opti
net = Net(input_dim, output_dim)
optimizer = optim.Adam(net.parameters(), lr=lrate)
for epoch in range(epochs):
    total_nll = 0
    for batch in batchize(train_d      ze):
        optimizer.zero_grad() #ze
        vecto                  _          ch, feat_index, label
        feat_v                            vectorized)
        label_vec = map(item              ectorized)
        feat_list = list(feat_
        label_list = list(lab  _
        x = torch.Tensor(feat_list)
        y = torch.LongTensor(label_list)
        loss = net.xtropy_loss(x,y)
        total_nll += loss
        loss.backward()
        optimizer.step()
    torch.save(net.state_dict(), net_path)
```

Sparse and Dense embeddings as input to ne

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Input to feedforward neural networks

- Assuming a bag-o $\boldsymbol{x}$ is the count of each word (
- To compute the hidden unit z_k:

$$z_k = \sum_{j=1}^{V} \theta_{j,}^{x}$$

- The connecti n units z_k form a vector e embedding of word j.
- If there is a lot of training data, word embeddi learned within the same network as the cl
- Word embeddings can also be learned separately from unlabeled data, using techniques such as Word2Vec and GLOVE.
- The latest trend is to learn *contextualized* word embeddings which are computed dynamically for each classification instance (e.g., ELMO, BERT). The requires more advanced architectures (Transformers) that we will talk about later in

One-hot encodings for features

A *one-hot* encoding is one in which each dimension corresponds to a unique feature, and the resulting feature vector of a c instance can be thought of as the sum of indicator feat in which a single dimension has a value of one while all others have a value of zero.

Example

When considering a bag-of-words represen lary of 40000 words. A short document of 20 words will be represented with a very sparse 40000-dimensional vector in which **at most** 20 dimensions have non-zero values

Combination of sparse vectors

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$+$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Sparse vs Dense representations

- Sparse representation
  - Each feature is a sparse vector in which one dimension is 1 and the rest are 0s (thus "one-hot").
  - Dimensionality of one-hot vector is same as number of distinct features
  - Features can be completely independent
    feature "
    as it is to "w
  - Feature
    summation.
- Dense representation
  - Each feature is a d-dimensional vector, with a d that is generally much shorter than that of a one-hot vector.
  - Model training will cause similar features to have similar vectors - information is shared between similar features.
  - Features can be combined via summation (or averaging), concatenation, or some combination of the two.
    - Concatenation if we care about relative position.

Using dense vectors in a classifier

Each core feature is embedded into a d dimensional space (typically 50-300), and represented as a vector in that space.

- Extract a set of core linguistic features that are relevant for predicting the output class

- For each feature onding vector v_i.

- Combine the vectors (either by concate on, or a combination of both) into an input vecto

  - Note: concatenation doesn't work for variable-length vectors such as document classification
- Feed x into a nonlinear classifier (feed-forward neural network).

Relationship between one-hot and dense vectors

- Dense represent
  pre-trained word embeddings
- One-hot and dense representations may not be as different as one might think
- In fact, using sparse, one-hot vectors as input
  neural netwo e
  network to lea
  feature] based on training data.
- With task-specific word embedding, t
  typically smaller, but the training objective for the embedding and the task objective are one and same
- With pre-trained word embeddings, the training data is easy to come by (just unannotated text), but the embedding object and task objective may diverge.

Two approaches of getting dense word vectors

- Count based methods, known in NLP as Distr
  Semantic Models (DSM) or Vector Seman
- Predictive m ork
  community entations
  for words, com
  - Distributed word representations duct of
    neural language models and later bec
    its own

Distributional semantics

- Based on the well-known observation of Z. Harris: Words are similar if they occur in the same context (Harris, 1954)
- Further summarized as a slogan: "You shall know a word by the company it keeps." (J. R. Firth, 1957)
- A long history o nt word meanin represents a context word it can occur wit
- Each word is represented as a sparse vecto high-dimensional space
- Then word distances and similarities can be computed with such a matrix

Steps for building a distributional semantic model

- Preprocess a (large) corpus: tokenization at a minimum, possibly lemmatization, POS tagging, or s
- Define the "context" for a target term (word). The context can be a window centered on the ta terms that are s (subject-of, o
- Compute a term-context matrix where ds to a term and each column corresponds to a c the target term.
- Each target term is then represented with a high-dimensional vector of context terms.

Mathematical processing for building a DSM

- Weight the term- h metrics such as Positive Pointwise Mutual Information (PPMI) to correct frequency bias

$$PPMI(x, y) = max(\log \frac{\quad}{y})$$

- Its dimension ation techniques such as *singular value d* (SVD)

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}$$

$$\boldsymbol{A} \in \mathbb{R}^{m \times n}, \boldsymbol{U} \in \mathbb{R}^{m \times k}, \boldsymbol{\Sigma} \in \mathbb{R}^{k \times k}, \boldsymbol{V} \in \mathbb{R}^{n \times k}, n >> k$$

- This will result in a matrix that has much lower dimension but retains most of the information of the original matrix.

Predictive methods

- Learns word emb xt, using various language model objectives.
- Decide on the context window
- Define the objective function that is used to pr context words based on the target word or pre word based on c
- Train the neural networ
- The resulting weight matrix will serve as t representation for the target word
- "Don't count, predict!" (Baroni et al, 2014) conducted systematic studies and found predict-based word embeddings outperform count-based embeddings.
- One of popular early word emdeddings are Word2vec embeddings.

Word2vec

- Word2vec is a software package that consists of two main models: CBOW (Continuous Bag of Words
- It popularized the use of distributed represe
to neural netw inspired
many follow LNet)
- It has it roots in language modeling (the use o
context to predict the target word), but gi
getting good language models and focus instead on getting good word embeddings.

Understanding word2vec: A simple CBOW model with only one context word in

- Input $\boldsymbol{x} \in \mathbb{R}^V$ and $x_{k'} = 0$ for $k' \neq k$. $\boldsymbol{\Theta} \in \mathbb{R}^{N \times V}$ is the weight matrix from the input layer to the hidden layer. Each column of $\boldsymbol{\Theta}$ is an N-dimensional vector representation ated word of the input layer.

- $\boldsymbol{\Theta'} \in \mathbb{R}^{V \times N}$ tput layer and $\boldsymbol{u}_{w_j}$ is the j-th row of score o_j for each target word w_j and context w omputed as:

$$o_j = \boldsymbol{u}_{w_j}^{\top} \boldsymbol{v}_{w_i}$$

- Finally we use softmax to obtain a posterior distribution

$$p(w_j|w_i) = y_j = \frac{exp(o_j)}{\sum_{j'=1}^{V} exp(o_{j'})}$$

where y_j is the output of the j-th unit in the output layer

Computing the hidden layer is just embedding lookup

Hidden layer computation:

$\boldsymbol{v}_{w_i} = \boldsymbol{z} = \boldsymbol{\Theta}\boldsymbol{x} =$

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.5 & 0.8 \\ 0.2 & 0.5 & 0.8 \end{bmatrix} \begin{bmatrix} 0.6 \\ 0.1 \\ 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix}$$

Note there is no activation at the hidden layer (or there is a linear activation function), so this is a "degenerate neural network".

Computing the output layer

$$\boldsymbol{o} = \Theta' \boldsymbol{z} = \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \end{bmatrix} \begin{bmatrix} 0.2 & 0.4 \\ 0.3 & 0.2 \\ 0.3 & 0.4 \\ 0.3 & 0.5 \end{bmatrix} \begin{bmatrix} 0.6 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.91 \\ 0.97 \\ 0.82 \\ 1.18 \\ 0.31 \\ 1.06 \\ 0.39 \\ 0.95 \\ 0.63 \end{bmatrix}$$

Each row of Θ' correspond to vector for a target word w_j.

Taking the softmax over the output

$$\begin{bmatrix} 0.11039215 \\ 0. \\ 0. \end{bmatrix}$$

$$\begin{bmatrix} 0.11039215 \\ 0.08016116 \end{bmatrix}$$

The output $\boldsymbol{y}$ is a probabilistic distribution over the entire vocabulary.

Input vector and output vector

Since there is no activation function at the hidden layer, the output is really just the dot product of the vector of the in context word and the vector of the output target wor

$$p(w_j|w_i) = y_j = \frac{\exp(e_j)}{\sum_{j'=1}^{V} \exp(e_{j'})} = \frac{\exp(\boldsymbol{u}_{w_j}^{\top} \boldsymbol{v}_{w_i})}{\sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^{\top} \boldsymbol{v}_{w_i})}$$

where $\boldsymbol{v}_{w_i}$ from $\boldsymbol{\Theta}$ is the **input vector** for word w_i and $\boldsymbol{u}_{w_j}$ from $\boldsymbol{\Theta}'$ is the **output vector** for word w_j

Computing the gradient on the hidden-output weights

- Use the familiar cro

$$\mathcal{L} = -\sum_{j}^{|V|} t_j \log y_j = -\log y_{j^\star}$$

where $j^\star$ is the index of the target word

- Given y_j is th t on the output is:

$$\frac{\partial \mathcal{L}}{\partial o_j} = y_j - t_j$$

$$\frac{\partial \mathcal{L}}{\partial \theta'_{ji}} = \frac{\partial \mathcal{L}}{\partial o_j} \frac{\partial o_j}{\partial \theta'_{ji}} = (y_j - t_j) z_i$$

- Update the hidden→output weights

$$\theta'_{ji} = \theta'_{ji} - \eta (y_j - t_j) z_i$$

Updating input→hidden weights

- Compute the erro

$$\frac{\partial \mathcal{L}}{\partial z_i} = \sum_{j=1}^{V} \frac{\partial \mathcal{L}}{\partial o_j} \frac{\partial o_j}{\partial z_i} = \sum_{j=1}^{V} (y_j - t_j)\theta'_{ij}$$

- Since

$$V$$

The derivative of $\mathcal{L}$ on the input ts:

$$\frac{\partial \mathcal{L}}{\partial \theta_{ik}} = \frac{\partial \mathcal{L}}{\partial z_i} \frac{z_i}{\theta_{ik}} = \sum_{j=1} (y_j - t_j)\theta'_{ji} x_k$$

- Update the input→hidden weights

$$\theta_{ki} = \theta_{ki} - \eta \sum_{j=1}^{V} (y_j - t_j)\theta'_{ij} x_k$$

Gradient computation in matrix form

$$\boldsymbol{D_o} = \boldsymbol{Y} - \quad = \begin{bmatrix} 0.11039215 \\ 0.10006362 \\ 0.11262222 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 39215 \\ 06362 \\ 37778 \\ 0.09693485 \\ 0.13893957 \\ 5820895 \\ 2312834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}$$

Computing the errors at the hidden layer

$$\boldsymbol{D_z} = \boldsymbol{D_o}^{\top}\boldsymbol{\Theta'} =$$

$$\begin{bmatrix} 0.110 & 0.106 & -0.887 & 0.097 & 0.139 & 0.0 & -0.110 & 0.080 \end{bmatrix}$$

$$\times \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix}$$

$$= \begin{bmatrix} -0.115 & 0.157 \end{bmatrix}$$

Computing the updates to Θ'

$$\nabla_{\Theta'} = \boldsymbol{D}_{\boldsymbol{o}}\boldsymbol{z}^{\top} = \begin{bmatrix} 0.410 \\ 0.106 \\ -0.887 \\ 0.097 \\ 0.139 \\ 0.058 \\ 0.123 \\ 0.063 \\ 0.110 \\ 0.080 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.8 & 0.8 \end{bmatrix} = \begin{bmatrix} 0.205 & 0.328 & 0.328 \\ 0.053 & 0.085 & 0.085 \\ -0.443 & -0.710 & -0.710 \\ 0.048 & 0.078 & 0.078 \\ 0.070 & 0.111 & 0.111 \\ 0.029 & 0.047 & 0.047 \\ 0.062 & 0.099 & 0.099 \\ 0.032 & 0.050 & 0.050 \\ 0.055 & 0.088 & 0.088 \\ 0.040 & 0.064 & 0.064 \end{bmatrix}$$

Computing the update to Θ

$$\nabla_\Theta = \boldsymbol{xD} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} -0.11538456 & 0 & 388611 \end{bmatrix}$$

$$= \begin{bmatrix} 0. & 0. & 0. \\ 0. & 0. & 0. \\ -0.11538456 & 0.15687943 & -0.19388611 \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \end{bmatrix}$$

CBOW for multiple context words

$$\boldsymbol{z} = \frac{1}{M}\boldsymbol{\Theta}(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \cdots \boldsymbol{x}_M)$$

$$= \frac{1}{M}(\boldsymbol{v}_{w_1} + \boldsymbol{v}_{w_2} + \cdot$$

where M is the number of words in the context, $_2, \cdots, w_M$ are the words in the con r. The loss function is

$$E = -\log p(w_j | w_1, \quad _2 \qquad _M$$

$$= -o_{j^\star} + \log \sum_{j'=1}^{V} \exp(o_{j'})$$

$$= -\boldsymbol{u}_{w_j}^\top \boldsymbol{z} + \log \sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^\top \boldsymbol{z})$$

Computing the hidden layer for multiple context words

$$\boldsymbol{z} = \boldsymbol{\Theta x} = \begin{bmatrix} 0.1 & 0.3 & 0.5 & & & & \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 \end{bmatrix} \begin{bmatrix} 0.6 \\ 1 \\ 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.5 \\ 3.0 \\ 3.0 \end{bmatrix}$$

During backprop, update vectors for four words instead of just one.

Skip-gram: model

where $w_{c,j}$ is the [redacted] tput layer, $w_{O,c}$ is the actual [redacted] w_I is the only input word, $y_{c,j}$ is the output of the [redacted] e c-th panel of the output layer; $o_{c,j}$ is the net i[redacted] j-th unit on the c-th panel of the output layer.

$$o_{c,j} = o_j = \boldsymbol{u}_{w_j} \cdot \boldsymbol{z}, \; for \; c = 1, 2, \cdots, C$$

Skip-gram: loss function

$$\mathcal{L} = -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C} \mid w_I)$$

$$= -\log \prod_{c=1}^{C} \frac{\exp(u_{j_c^\star})}{\sum_{j'=1}^{V} \exp(u_{j'})}$$

$$= -\sum_{c=1}^{C} u_{j_c^\star} + C \cdot \log \sum_{j'=1}^{V} \exp(u_{j'})$$

where $j_c^\star$ is the index of the actual c-th output context word.

Combine the loss of C context words with multiplication. Note: $o_{j'}$ is the same for all C panels

Skip-gram: updating the weights

- We take the derivative of the output of every unit on every position, $o_{c,j}$, and obtain

$$e_{c,j} = \frac{\partial \mathcal{L}}{\partial o_{c,j}} = y_{c,j} - t_{c,j}$$

which is the prediction error of the unit.

- We define an E_V-dimensional vector E_j as the sum of the prediction errors:

$$E_j = \sum_{c=1}^{C} e_{c,j}$$

$$\frac{\partial \mathcal{L}}{\partial \theta'_{ji}} = \sum_{c=1}^{C} \frac{\partial \mathcal{L}}{\partial o_{c,j}} \cdot \frac{\partial o_{c,j}}{\partial \theta'_{ji}} = E_j \cdot z_i$$

- Updating the hidden→output weight matrix:

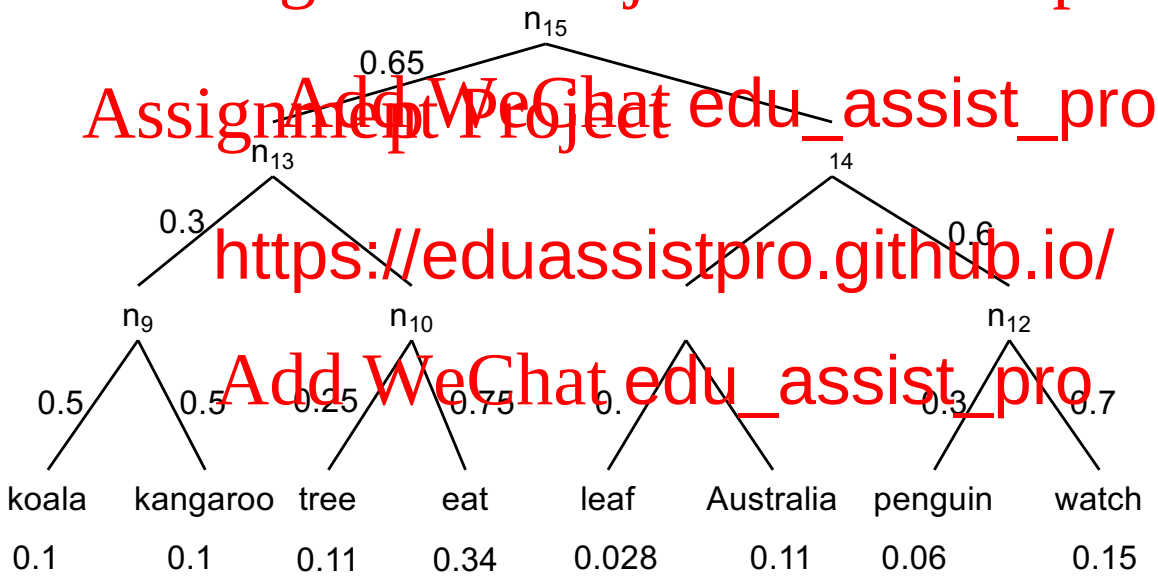
$$\theta'_{ji} = \theta'_{ji} - \eta \cdot E_j \cdot z_i$$

- No change in how the input→hidden weights are updated.

Optimizing computational efficiency

- Computing softmax at the output layer is exp
  involves iterative over the entire vocabular
- Two methods for optimizing computational efficiency
  - **Hierar** ute the probabi plexity from $|V|$ to $\log|V|$.
  - **Negative sampling**: Instead of ap r all the words in the vocabulary, only sample a small number of words that are not actual context words in the training corpus

Hierarchical softmax



Computing the probabilities of the leaf nodes

$$P(\text{“}Kangaroo\text{”}|z) = P_{n_0}(Left|\boldsymbol{z}) \times P_{n_0}(Right|\boldsymbol{z})$$

$$P_n(Right|\ ) = 1$$

$$P_n(Left|z) = \sigma(\gamma$$

where γ_n is a vector from a set of new parameters that replace $\boldsymbol{\Theta}$

Huffman Tree Building

A simple algorithm:

- Prepare a collection of n initial Huff ich is a single leaf node. Put the n tre organized by weight (frequency).
- Remove the fir oin these two tree trees as children, and whose weight is the su of the two children trees. Put this new tree in queue.
- Repeat steps 2-3 until all of the partial Huffman trees have been combined into one.

Negative sampling

- Computing softmax over the vocabulary is expensive. Another alternative is to approximate softmax by only updating a small sample of (context) words at a time.
- Given a pair of words (w, c), let P e the probability of t us, and $P(D =$ ot come from the corpus.
- This probability can be modeled as a sigm

$$P(D = 1|w, c) = \sigma(\boldsymbol{u}_w{}^\top \boldsymbol{v}_c) = \frac{1}{1 + e^{\boldsymbol{u}_w{}^\top \boldsymbol{v}_c}}$$

New learning objective for negative sampling

- We need a new objec
  minimize the follo

$$\mathcal{L} = -\sum_{w_j \in D} \log \sigma(o_{w_j}) - \log \sigma(-o_{w_j})$$

where D is a set of correct context - target wo D' is a set of incorre

- Note that we us
  negative samples. In the skip-gram algo
  multiple positive context words in the ou
  algorithm, there will be only one positive target word.

- The derivative of the loss function with respect to the output word will be:

$$\frac{\partial \mathcal{L}}{\partial o_{w_j}} = \sigma(o_{w_j}) - t_{w_j}$$

where $t_{w_j} = 1$ if $w_j \in D$ and $t_{w_j} = 0$ if $w_j \in D'$

Updates to the hidden output weights

- Compute the gradient on the output weight $\frac{\partial}{\partial}$

- When updating the output weights, onl
  for words in the positive sample and negat
  be updated:

Updates to the input $\rightarrow$ hidden weights

- Computing the derivative of the loss function with respect to the hidden layer

$$\frac{\partial \mathcal{L}}{\partial} = \qquad (\sigma(o_w \qquad w \quad w_j$$

- In the CBOW algorithm, the weights for a words will be updated. In the Skip-gram all t for the target word will be updated.

$$\boldsymbol{v}_{w_i} = \boldsymbol{v}_{w_i} - \eta(\sigma(o_{w_j}) - t_{w_j})\boldsymbol{u}_{w_j} x_i$$

How to pick the negative samples?

- If we just randomly p
  of any given word w_i getting picked is:

$$p(w_i) = \frac{freq(w_i)}{V}$$

  More frequen
  may not be ideal

- Adjust the formula to give the less frequen
  chance to get picked:

$$p(w_i) = \frac{freq(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{V} freq(w_j)^{\frac{3}{4}}}$$

- Generate a sequence of words using the adjusted probability, and randomly pick $n_{D'}$ words

Use of embeddings: word and short document similarity

- Word embeddings can be used to compute word similarity with cosine similarity

$$sim_{cos}(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{|\boldsymbol{u}||\boldsymbol{v}|}$$

- How accurat ity can also be used t
- They can also be used to compute the simila documents

$$sim_{doc}(D_1, D_2) = \sum_{i=1}^{m} \sum_{j=1}^{n} cos(\boldsymbol{w}_i^1, \boldsymbol{w}_j^2)$$

Use of embeddings: word analogy

- What's even more impressive is that they can be used to compute word analogy

$$analogy(m : w \quad k :?) = \text{argmax} \qquad \boldsymbol{m} + \boldsymbol{w})$$

$$analogy(m \qquad \boldsymbol{v}, \boldsymbol{k}) = cos(\boldsymbol{v}, \boldsymbol{m})$$

$$v \in$$

$$+ \epsilon$$

$$analogy(m : w \rightarrow k :?) = \underset{v \in V \setminus m,k,w}{\text{argmax}} \frac{cos(\boldsymbol{v}, \boldsymbol{k})cos(\boldsymbol{v}, \boldsymbol{w})}{cos(\boldsymbol{v}, \boldsymbol{m}) + \epsilon}$$

Word analogy

Use of word embeddings

- Computing word similarities is not a "real" problem in the eyes of many

- The most important use word embeddings is predict the outcome of tasks that have real-w

- Many follow-embeddings

  - word2vec: http://vectors.nlpl.e
  - fasttext: https://fasttext.cc/do ctors.html

  - GLOVE: https://nlp.stanford.e e
- Contextualized word embeddings:
  - https://allennlp.org/elmo
  - BERT: https://github.com/google-research/bert
  - Roberta: https://pytorch.org/hub/pytorch_fairseq_roberta

Embeddings in Pytorch

Commonly used neural architectures

- One important aspect of neural network mo
  out the right representation of a problem
- Commonly u
  - Variant
    been inst
  - Convolutional Networks (CNN)
    in image processing and some NLP pro
    classification)

hich have
LP
ective
e

Convolution

Convolutional Networks

- A convolutional network is designed to identify indicative local indicators in a large structure, and combine them to produce a **fixed size** vector representation of the st
pooling function, capturing the local aspect
informative o
- A convolutio
feedforward network is.
- It has been tremendously successful in im
computer vision), where the input is the raw pixels of an image
- In NLP, it has been shown to be effective in sentence classification, etc.

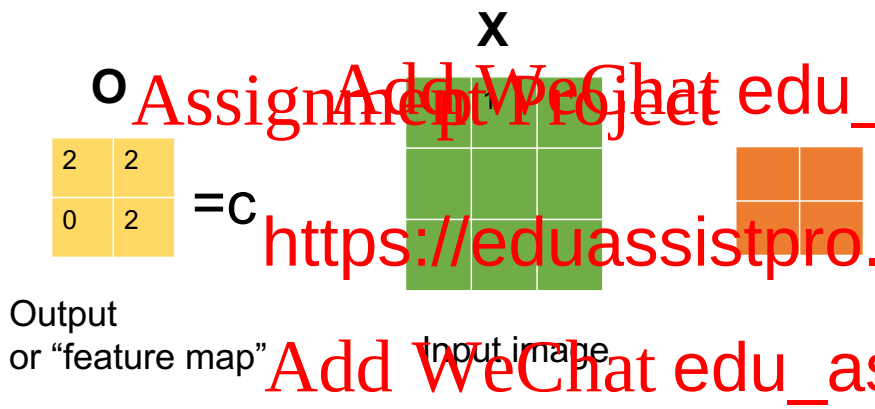
Why it has been so effective in image processing

Image pixels

Four operations in a convolutional network

- Convolution
- Non-linear ac
- Pooling or sub
- Classification with a fully connected lay

Image convolution



Kernel size = (2,2), strides = 1

Image convolution

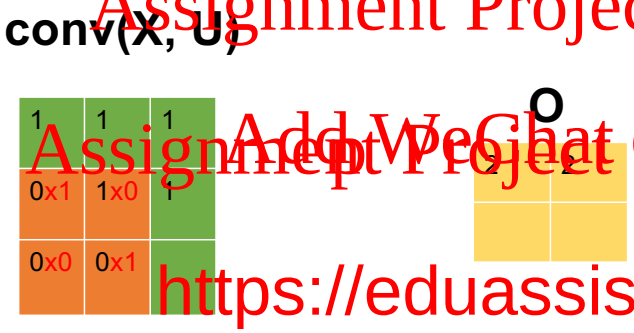
conv(X, U)

Kernel size = (2,2), strides = 1

Image convolution

Kernel size = (2,2), strides = 1

Image convolution



Kernel size = (2,2), strides = 1

Image convolution

conv(X, U)

Kernel size = (2,2), strides = 1

Forward computation

$$o_{11} = x_{11}u_{11} + x_{12}u_{12} + x$$

$$o_{12}$$ $$23 u_{22}$$

$$o_{21}$$ $$21 \quad 11 \qquad 22 \quad 12 \qquad 31 \quad 21 \qquad 32 u_{22}$$

$$o_{22} = x_{22}u_{11} + x_{23}u_{12}$$ $$22$$

ReLU

- Nonlinear transformation with ReLU

$$\max(0, input)$$

- As we know, ReLU at does not change the dimension of the feat
- ReLU replaces all negative pixel values in t h 0

Image ReLU

ReLU activation and Max pooling

- ReLU activation is a component-wise function and does not change the dimension of the input

$$\begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix}$$

- Max pooling does change the dimension specify the pool size and strides

$$[2] = Max\left(\begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix}\right)$$

$$\text{pool size} = (2,2), \text{strides} = 2$$

Training a CNN

- Loss functions: Cross entropy loss, square
- What are the parameters of a CNN?
  - The filter ard networ ses
- Computing the gradient for the convolu nt from a feedforward neural network...

Computing the gradient on U

$$\frac{\partial E}{\partial u_{11}} = \frac{\partial E}{\partial o_{11}}\frac{\partial o_{11}}{\partial u_{11}} + \frac{\partial E}{\partial o_{12}}\frac{\partial o_{12}}{\partial u_{11}} + \frac{\partial E}{\partial o_{21}}\frac{\partial o_{21}}{\partial u_{11}} + \frac{\partial E}{\partial o_{22}}\frac{\partial o_{22}}{\partial u_{11}}$$
$$= \frac{\partial E}{\partial o_{11}}x_{11} + \frac{\partial E}{\partial o_{12}}x_{12} + \frac{\partial E}{\partial o_{21}}x_{21} + \frac{\partial E}{\partial o_{22}}x_{22}$$

$$\frac{\partial E}{\partial u_{12}} = \frac{\partial E}{\partial o_{11}}\frac{\partial o_{11}}{\partial u_{12}} + \frac{\partial E}{\partial o_{12}}\frac{\partial o_{12}}{\partial u_{12}} + \frac{\partial E}{\partial o_{22}}\frac{\partial o_{22}}{\partial u_{12}}$$

$$\frac{\partial E}{\partial u_{21}} = \frac{\partial E}{\partial o_{11}}\frac{\partial o_{11}}{\partial u_{21}} + \frac{\partial E}{\partial o_{12}}\frac{\partial o_{12}}{\partial u_{21}} + \frac{\partial E}{\partial o_{22}}\frac{\partial o_{22}}{\partial u_{21}}$$
$$= \frac{\partial E}{\partial o_{11}}x_{21} + \frac{\partial E}{\partial o_{12}}x_{22} + \frac{\partial E}{\partial o_{21}}x_{31} + \frac{\partial E}{\partial o_{22}}x_{32}$$

$$\frac{\partial E}{\partial u_{22}} = \frac{\partial E}{\partial o_{11}}\frac{\partial o_{11}}{\partial u_{22}} + \frac{\partial E}{\partial o_{12}}\frac{\partial o_{12}}{\partial u_{22}} + \frac{\partial E}{\partial o_{21}}\frac{\partial o_{21}}{\partial u_{22}} + \frac{\partial E}{\partial o_{22}}\frac{\partial o_{22}}{\partial u_{22}}$$
$$= \frac{\partial E}{\partial o_{22}}x_{11} + \frac{\partial E}{\partial o_{12}}x_{23} + \frac{\partial E}{\partial o_{21}}x_{32} + \frac{\partial E}{\partial o_{22}}x_{33}$$

Summing up errors from all outputs that the filter component has contributed to.

Reverse Convolution

The computation of the gradient on the filter can be ve
a reverse convolution:

$$\begin{bmatrix} \frac{\partial E}{\partial u_{11}} & \frac{\partial E}{\partial u_{12}} \\ \frac{\partial E}{\partial u_{21}} & \frac{\partial E}{\partial u_{22}} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} * \begin{bmatrix} \frac{\partial E}{\partial o_{11}} & \frac{\partial E}{\partial o_{12}} \\ \frac{\partial E}{\partial o_{21}} & \frac{\partial E}{\partial o_{22}} \end{bmatrix}$$

Computing the gradient on X (if this is not the input layer)

$$\frac{\partial E}{\partial x_{11}} = \frac{\partial E}{\partial o_{11}} u_{11} + \frac{\partial E}{\partial o_{12}} u_{12} + \frac{\partial E}{\partial o_{21}} u_{21} + \frac{\partial E}{\partial o_{22}} u_{22}$$

$$\frac{\partial E}{\partial x_{12}} = \frac{\partial E}{\partial o_{11}} u_{12} + \frac{\partial E}{\partial o_{12}} u_{11} + \frac{\partial E}{\partial o_{21}} u_{22} + \frac{\partial E}{\partial o_{22}} u_{21}$$

$$\frac{\partial E}{\partial x_{13}} = \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} u_{12} + \frac{\partial E}{\partial o_{21}} u_{22} + \frac{\partial E}{\partial o_{22}} u_{21}$$

$$\frac{\partial E}{\partial x_{21}} = \frac{\partial E}{\partial o_{11}} u_{21} + \frac{\partial E}{\partial o_{12}} u_{22} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} 0$$

$$\frac{\partial E}{\partial x_{22}} = \frac{\partial E}{\partial o_{11}} u_{22} + \frac{\partial E}{\partial o_{12}} u_{21} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} 0$$

$$\frac{\partial E}{\partial x_{23}} = \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} u_{22} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} u_{21}$$

$$\frac{\partial E}{\partial x_{31}} = \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} u21 + \frac{\partial E}{\partial o_{22}} 0$$

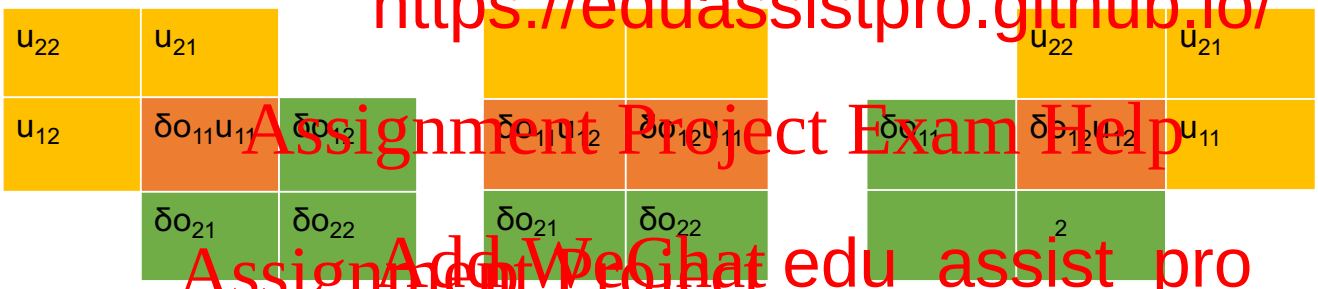
$$\frac{\partial E}{\partial x_{32}} = \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} u_{22} + \frac{\partial E}{\partial o_{22}} u_{21}$$

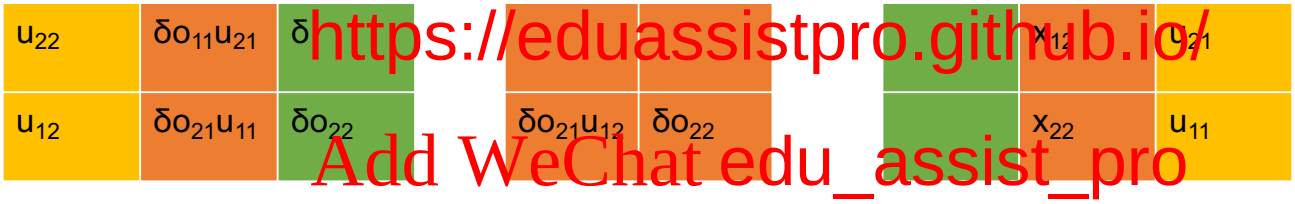
$$\frac{\partial E}{\partial x_{12}} = \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} u_{22}$$

Full convolution

$$\left(\begin{bmatrix} \frac{\partial E}{\partial x_{11}} & \frac{\partial E}{\partial x_{12}} & \\ \frac{\partial E}{\partial x_{21}} & \frac{\partial E}{\partial x_{22}} & \\ \frac{\partial E}{\partial x_{31}} & \frac{\partial E}{\partial x_{32}} & \frac{\partial E}{\partial u_{33}} \end{bmatrix}, \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \right)$$

Gradient on X if it is not the inputs





Sample code of 2D convolution with Keras


```
model = Sequential()
model.add(Conv2D(32, kernel_size                    (1, 1),
                 activation='relu
                 input_shape=inp  _
model.add(MaxP                   _                  ides=(2, 2)))
model.add(Conv
model.add(MaxP                   _
model.add(Flatten())
model.add(Dense(1000, activation='rel                  ))
model.add(Dense(num_classes, activ                     x'))
```


Tutorials on how to train CNNs can be found on the Tensorflow website:

https://www.tensorflow.org/tutorials/images/cnn

Why convolutational networks for NLP?

- Even though bag-of-word models are simpl
  some text classification tasks, they don't acc
  where multipl s
  "not interesti
- The analogy with image processing is if th
  as separate features. (The analogy migh

Input to a convolutional network in a text classification task

The input to a convolutional network can be pretrained word embeddings (e.g., the weight matrix produced by GLOVE[1]) and the input s e:

$$\boldsymbol{X}^{(0)} = \boldsymbol{\Theta}_{[w_1,\ldots,w_M]} \in \mathbb{R}^{K_e \times M}$$

where $\boldsymbol{e}_{w_m}$ is a column vector of zeros, with a 1 at position w_m, K_e is the size of embeddings

[1] https://nlp.stanford.edu/projects/glove

Alternative text representations

- Alternatively, a text can be represented as a sequence of word tokens $w_1, w_2, w_3, \cdots, w_M$. This view is useful for models such as **Convolutional Neural Networks**, or **convnets**, which processes text as a sequence.
- Each word token is represented by a one-hot vector $\boldsymbol{e}_{w_m}$, with dimension V. The entire document is represented by the horizontal concatenation of these one-hot vectors: $\boldsymbol{W} = [\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \cdots, \boldsymbol{e}_{w_M}]$.
- To show that this is equivalent to the bag-of-words model, we can recover the word count from the matrix-vector product $\boldsymbol{W}[1, 1, \cdots, 1]^\top \in R^V$.

"Convolve" the input with a set of filters

- A *filter* is a weight matrix of dimension $\boldsymbol{C}^{(k)} \in \mathbb{R}^{K_e \times h}$ where $\boldsymbol{C}^{(k)}$ is the kth filter. Note the first dimension of the filter is the same as the size of the embedding.
  - Unlike image processing, the filter doesn't full width
- To merge adja by sliding a set of filters across it:

$$\boldsymbol{X}^{(1)} = f(\boldsymbol{b}$$

where f is an activation function (e.g., tanh, ReLU), b is a vector of bias terms, and $*$ is the convolution operator.

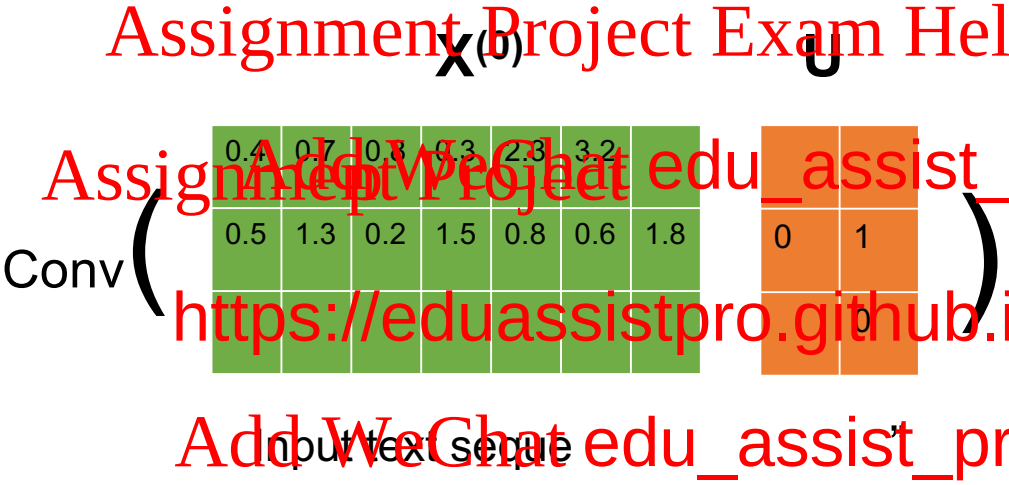
Computing the convolution

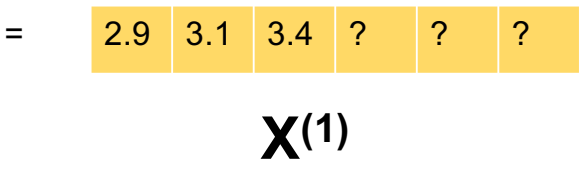
- At each position m, we compute the element-wise product of the kth filter and the sequence of words of window size h (think of it as an ngram of length h) starting a m and take it $(k) \odot \boldsymbol{X}^{(0)}_{m:m+h-1}$
- The value of the mth position with the kth filter is computed as:

$$x^{(1)}_{m,k} = f\left(b_k + \sum_{k'=1}^{K_e} \sum_{n=1}^{h} \right)$$

- When we finish the convolution step, if we have K_f filters of dimension $\mathbb{R}^{K_e \times h}$, then $\boldsymbol{X}^{(1)} \in \mathbb{R}^{K_f \times M-h+1}$
- In practice, filters of different sizes are often used to captured ngrams of different lengths, so $\boldsymbol{X}^{(1)}$ will be K_f vectors of variable lengths, and we can write the size of each vector of h_k

Convolution step when processing text





Padding

- To deal with the beginning and end of the input, t matrix is often padded with h 1 c os at the beginning. **nvolution**
- If no padding is a n layer will be $h - 1$ units s r than the input. This is known as **narrow convolution**.

Pooling

- After D convolutional layers, assuming filters have identical lengths, we have a representation of the docu $\boldsymbol{X}^{(D)} \in \mathbb{R}^{L \times M}$.

- It is very likely th , so we need to tur before feeding them to a feedward ne m classification.

- This can done by **pooling** across times (over the sequence of words)

Prediction and training with CNN

- The CNN needs to be fed into a feedforward network to make a prediction $\hat{y}$ and compute the loss $\ell^{(i)}$ in training.
- Parameters of a CNN includes the weight ma feedforward network and the filters , s well as the biases.
- The paramet , which may involve computing the gradient for t function.

$$\frac{\partial z_k}{\partial x_{k,m}^{(D)}} = \begin{cases} 1, & x_{k,m}^{(D)} = \max\left(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \cdots, x_{k,M}^{(D)}\right) \\ 0, & \text{Otherwise} \end{cases}$$

Different pooling methods

- Max pooling

$$z_k = \max\left(x_{k,1}^{(B)}, x_{k,2}^{(D} \ldots x_{k,M}\right)$$

- Average pooli

$$z_k = \frac{1}{M} \sum_{m=1} x_{k,m}$$

A graphic representation of a CNN

Figure 1: Caption

Sample code of convolution with Keras


```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, \\
    Embedding, Flatten, Conv1D, GlobalMaxPooling1D

model = Sequential()
model.add(Embedding(input_dim=n_input_words,
                    _                  g_dim,
                    _                  n))
model.add(Conv1D(num_filters, ker    _        \\
                 activation='relu'))
model.add(GlobalMaxPooling1D())
model.add(Dense(50, activation='relu'))
model.add(Dense(3, activation='softmax'))
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

An Interim Summary of Supervised Learning Methods

- In all the linear and no
  far, we assume we have labeled training data where we can perform supervised learning:
  - a **training set** where you get observa bels y;
  - a test set where you only get observat
- A summary of t
  discussed so fa
  - Linear m erceptron, Support Vector Machines
  - Non-linear models: feed-forward n ional Networks
  - Sparse vs dense feature representations as input to classifiers
- Given sufficient amounts of high-quality data, supervised learning methods tend to produce more accurate classifiers than alternative learning paradigms

NLP problems that can be formulated as simple text classifications

- An NLP problem can be formulated as a simple text classification if there is no inter-dependenc
labels (y) of the classification instances:
  - Word se
  - Sentim
  - Genre cla
  - Others
- NLP problems that cannot be formulat
classifications (or you can, but the results won't be optimal)
  - Sequence labeling problems such as POS tagging, Named Entity Recognition
  - Structured prediction problems such as syntactic parsing

Beyond Supervised Learning

There are other learning scenarios where labeled training sets are available to various degree or not available at all

- When there is no labeled data at all, we'll have to d **unsupervis** ring, variants of the EM algorit
- When there is a small amount of labeled dat to try **semi-supervised learning**
- When there is a lot of labeled data in one domain but there is only a small of labeled data in the target domain, we might try **domain adaptation**

K-Means clustering algorithm

K-means clustering algorithm

procedure K-MEANS($\boldsymbol{x}_{1:N}, K$)

for $i \in 1 \cdots N$ **do**

$z^{(i)} \leftarrow$ RANDOMINT$(1, K)$ ▷ mbership

repeat

for k ute cluster centers

$$\boldsymbol{v}_k \leftarrow \frac{1}{\sum_{i=1}^{N} \delta(z^{(i)}=k)} \sum_{i}^{N} \boldsymbol{x}^{(i)}$$

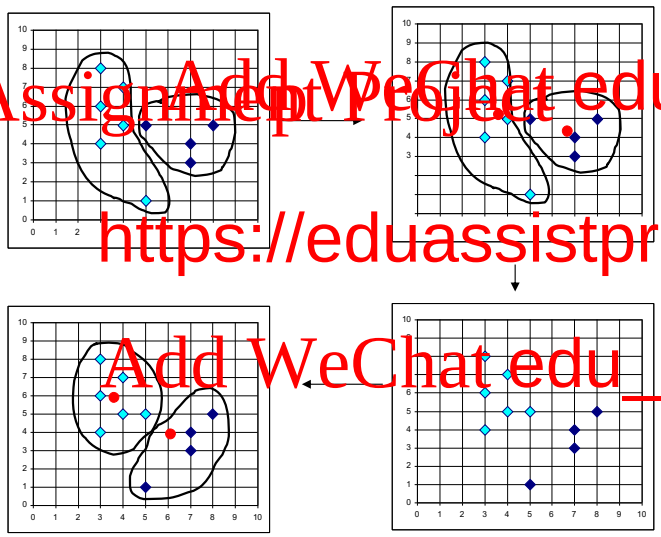
for $i \in 1 \cdots N$ **do**

$z^{(i)} \leftarrow \text{argmin}_K \, ||\boldsymbol{x}^{(i)} - \boldsymbol{v}_k||^2$

until Converged

return $z^{(i)}$

K-Means training



- K-means clustering is non-parametric and has no parameters to update
- The number of clusters need to be pre-specified before the training process starts

Semi-supervised learning

- Initialize parameters with supervised learning and then apply unsupervised learning (such as the EM algorithm)
- Multi-view learning: Co-training
  - divide features into multiple views, and tra
    each vie
  - Each cla d
    instances, using only the features ava
    predictions are then used as ground tr
    classifiers associated with the other vi
  - Named entity example: named entity view and local context view
  - Word sense disambiguation: local context view and global context view

Domain adaptation

Supervised domain adaptation (Daumé in

- Creates copies of each feature: one for each domain and one for the cross-domain setting

$$\boldsymbol{f}(\boldsymbol{x}, y, d) = \{(\text{boring, NEG, Movie}):1,$$
$$(\text{fl} \qquad \text{EG, *}):1,$$
$$(\text{t} \qquad \text{day-old, NEG, *}):1,$$
$$\ldots\}$$

where d is the domain.

- Let the learning algorithm allocate weights between domain specific features and cross-domain features: for words that facilitate prediction in both domains, the learner will use cross-domain features. For words that are only relevant to a particular domain, domain-specific features will be used.

Other learning paradigms

- Active learning: A learning that is often used to reduce the number of instances that have to be annotate produce the same level of accuracy
- Distant super generate som me external resource such as a dictionary. Fo generate named entity annotation wit
- Multitask learning: The learning induces a representation that can be used to solve multiple tasks (learning POS tagging with syntactic parsing)