Linear models: Recap

Linear models:

- Perceptron

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

- Naïve Bayes:

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

- 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 representation: $\boldsymbol{f}(\boldsymbol{x}, y)$

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

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

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

- 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 $K \times V$ matrix

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_V \end{bmatrix} \qquad \boldsymbol{\Theta} = \begin{array}{c} \\ y=1 \\ y=2 \\ \cdots \\ y=K \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & \cdots & x_V \end{array} \\ \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & \cdots & \theta_{1,V} \\ \theta_{2,1} & \theta_{2,2} & \cdots & \theta_{2,V} \\ \cdots & \cdots & \cdots & \cdots \\ \theta_{K,1} & \theta_{K,2} & \cdots & \theta_{K,V} \end{bmatrix} \end{array}$$

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

Scores for each class

- Verify that ψ_1, ψ_2, $\cdots$, ψ_K correspond to the scores for each class

$$\boldsymbol{\Psi} = \boldsymbol{\Theta}\boldsymbol{x} = \begin{bmatrix} \boldsymbol{\theta}_1 \cdot \boldsymbol{x} = \psi_1 \\ \boldsymbol{\theta}_2 \cdot \boldsymbol{x} = \psi_2 \\ \cdots \\ \boldsymbol{\theta}_3 \cdot \boldsymbol{x} = \psi_K \end{bmatrix}$$

Implementation in Pytorch

In [48]:

```
weights = torch.randn(3,9)
print(weights)
input = torch.randn(9)
print(input)
output = torch.matmul(weights, input)
print(output)
softmax = nn.Softmax(dim=0)
probs = softmax(output)
print(probs)
```

```
tensor([[-0.3518, -0.3291,  1.6937, -0.9947, -0.1390, -0.7788, -0.0252, -0.1557,
         -1.2138],
        [-1.2000,  1.3527,  1.9529, -1.3182,  0.1101, -0.7105, -0.4409,  0.9753,
         -0.0821],
        [-0.1561, -0.3144,  0.4833,  0.9997,  0.5840, -1.4633,  1.1353,  0.3069,
         -0.0584]])
tensor([ 0.0148,  0.0565, -0.6462, -0.0155, -0.5532, -0.8514, -0.1339,  0.5056,
         0.6025])
tensor([-1.1695, -0.1363,  0.5428])
tensor([0.1069, 0.3005, 0.5926])
```

Digression: Matrix multiplication

- Matrix with m rows and n columns:

$$A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}, C = AB \in \mathbb{R}^{m \times p}$$

where $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$

- Example:

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 7 \\ 1 & 2 & 4 \end{bmatrix}$$

Digression: 3-D matrix multiplication

```
In [27]: import torch
         input = torch.randint(5, (2, 3, 4))
         print(input)
         mat2 = torch.randint(5, (2, 4, 3))
         print(mat2)
         out = torch.bmm(input,mat2)
         print(out)
```

```
tensor([[[0, 1, 3, 2],
         [3, 1, 4, 1],
         [0, 1, 2, 0]],

        [[3, 2, 3, 4],
         [4, 3, 2, 4],
         [3, 0, 2, 3]]])
tensor([[[0, 2, 0],
         [4, 4, 2],
         [2, 2, 4],
         [1, 1, 4]],

        [[3, 0, 4],
         [3, 0, 1],
         [0, 0, 4],
         [2, 4, 2]]])
tensor([[[12, 12, 22],
         [13, 19, 22],
         [ 8,  8, 10]],

        [[23, 16, 34],
         [29, 16, 35],
         [15, 12, 26]]])
```


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

SoftMax

- SoftMax, also known as normalized exponential function.

$$\text{SoftMax}_i(\boldsymbol{\psi}) = \frac{\exp \psi_i}{\sum_j^K \exp \psi_j}$$

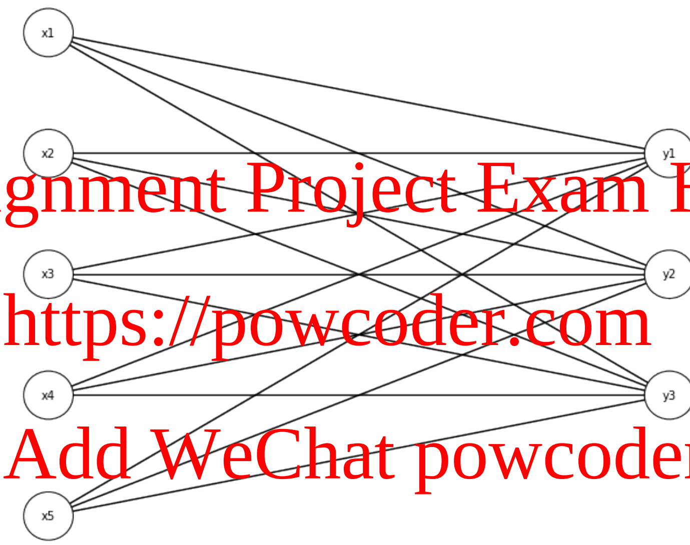
  for $i = 1, 2, \cdots, K$

- Applying SoftMax turns the scores into a probabilistic distribution:

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

- Verify this is exactly logistic regression

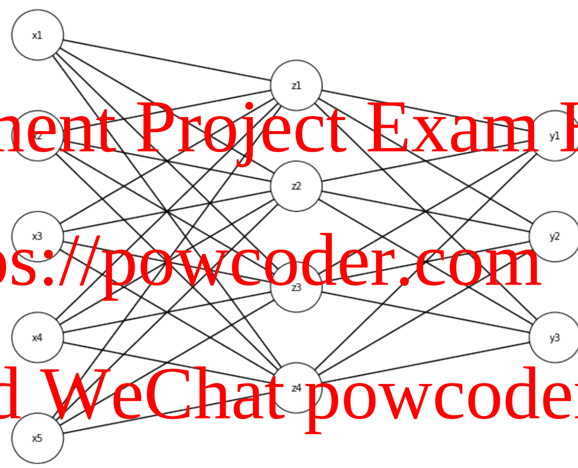
Logistic regression as a neural network



$$\boldsymbol{y} = \mathrm{SoftMax}(\boldsymbol{\Theta x})$$
$$V = 5 \; K = 3$$

Going deep

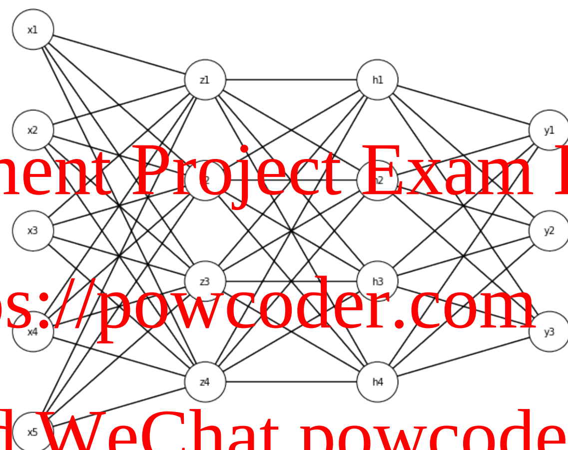
- There is no reason why we can't add layers in the middle



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

Going even deeper

- There is no reason why we can't add layers in the middle



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

- But why?

Non-linear classification

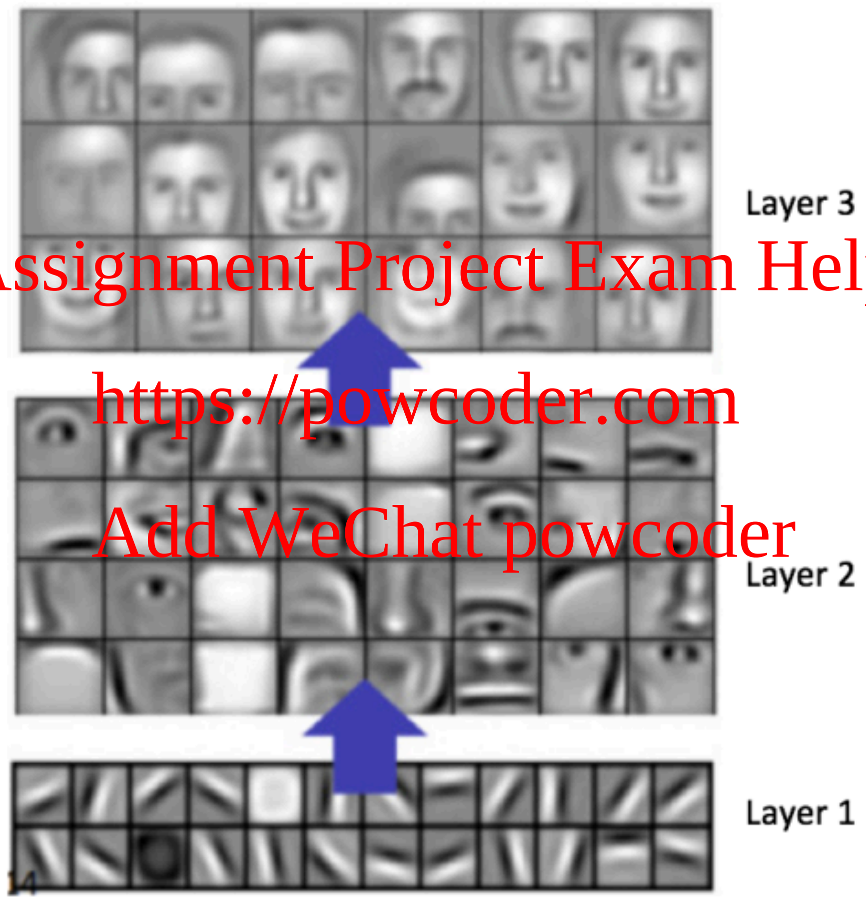
Linear models like Logistic regression can map data into a high-dimensional vector space, and they are expressive enough and work well for many NLP problems, why do we need more complex non-linear models?

- There have been rapid advances in **deep learning**, a family of nonlinear methods that learn complex functions of the input through multiple layers of computation.
- Deep learning facilitates the incorporation of **word embeddings**, which are dense vector representations of words, that can be learned from massive amounts of unlabeled data.
  - 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 imagine a scenario that you can predict the shapes of parts of an image, mouth, hand, ear.
- In text processing, we can imagine a similar scenario. Let's say we want to classify movie reviews (or movies themselves) into a label set of $\{Good, Bad, OK\}$. Instead predicting these labels directly, we first predict a set of composite features 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 $\boldsymbol{x}$ to predict the features $\boldsymbol{z}$.** Specifically, train a logistic regression classifier to compute $P(z_k|\boldsymbol{x})$, for each $k \in \{1, 2, \cdots, K_z\}$
- **Use the features $\boldsymbol{z}$ to predict the label y.** Train a logistic regression classifier to compute $P(y|\boldsymbol{z})$. $\boldsymbol{z}$ is unknown or hidden, so we will use the $P(\boldsymbol{z}|\boldsymbol{x})$ as the features.

Caveat: it's easy to demonstrate what this is what the model does 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:

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

- The weight matrix $\boldsymbol{\Theta}^{(x\rightarrow z)} \in \mathbb{R}^{k_z \times V}$ is constructed by stacking (not concatenating, as in linear models) the weight vectors for each z_k,
$\boldsymbol{\Theta}^{(x\rightarrow z)} = \left[\boldsymbol{\theta}_1^{x\rightarrow z}, \boldsymbol{\theta}_2^{x\rightarrow z}, \cdots, \boldsymbol{\theta}_{K_z}^{x\rightarrow z}\right]^\top$
- We assume an offset/bias term is included in $\boldsymbol{x}$ and its parameter is included in each $\boldsymbol{\theta}_k^{x\rightarrow z}$

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

Activation functions

- Sigmoid: The range of sigmoid function is $(0, 1)$.

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

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

- Tanh: The range of the tanh activation function is $(-1, 1)$

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

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

- 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.randn(4)
sigmoid = nn.Sigmoid()
output = sigmoid(input)

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

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

- The weight matrix $\boldsymbol{\Theta}^{(z \to y)} \in \mathbb{R}^{k_y \times k_z}$ again is constructed by stacking weight vectors for each y_k:
$\boldsymbol{\Theta}^{(z \to y)} = \left[\boldsymbol{\theta}_1^{z \to y}, \boldsymbol{\theta}_2^{z \to y}, \cdots, \boldsymbol{\theta}_{K_y}^{z \to y}\right]^\top$
- The vector of probabilities over each possible value of y is denoted:
$P(\boldsymbol{y}|\boldsymbol{z}; \boldsymbol{\Theta}^{(z \to y)}, \boldsymbol{b}) = \mathrm{SoftMax}(\boldsymbol{\Theta}^{(z \to y)}\boldsymbol{z} + \boldsymbol{b})$

The negative loglikelihood or cross-entropy loss

In a multi-class classification setting, a softmax output produces a probabilistic distribution over possible labels. It works well together with negative conditional likelihood (just like logistic regression)

$$-\mathcal{L} = -\sum_{i=1}^{N} \log P(y^{(i)} | \boldsymbol{x}^{(i)}; \boldsymbol{\Theta})$$

or **cross entropy loss**:

$$\tilde{y}_j \triangleq Pr(y = j | \boldsymbol{x}^{(i)}; \boldsymbol{\Theta})$$

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

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 (remember perceptron) with a margin loss:

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

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

Training a feedforward network with the backpropagation algorithm

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 that drives parameter estimation:

$$\begin{aligned}
\boldsymbol{z} &\leftarrow f\left(\boldsymbol{\Theta}^{x\to z}\boldsymbol{x}^{(i)}\right), && \boldsymbol{z} \in R^{k_z} \\
\tilde{\boldsymbol{y}} &\leftarrow \text{SoftMax}\left(\boldsymbol{\Theta}^{z\to y}\boldsymbol{z} + \boldsymbol{b}\right), && \tilde{\boldsymbol{y}} \in R^{k_y} \\
\ell^{(i)} &\leftarrow -\boldsymbol{e}_{y^{(i)}} \cdot \log \tilde{\boldsymbol{y}}
\end{aligned}$$

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

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

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

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

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 $\mathbf{\Theta}^{x\to z}$, and $\boldsymbol{\theta}_k^{z\to y}$ is the kth column of the matrix $\mathbf{\Theta}^{z\to y}$.

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

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

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

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

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:

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

where

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

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

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

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

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

One more step to compute the gradient of the loss with respect to the weights:

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

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

Similarly, we can also compute the derivative with respect to the 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 $\boldsymbol{\Theta}^{(x \to z)}$

Apply the old chain rule in differentiation:

$$
\begin{aligned}
\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x \to z)}} &= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial z_k}{\partial \theta_{n,k}^{(x \to z)}} \\
&= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial f\left(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}\right)}{\partial \theta_{n,k}^{(x \to z)}} \\
&= \frac{\partial \ell^{(i)}}{\partial z_k} f'\left(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}\right) x_n
\end{aligned}
$$

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.

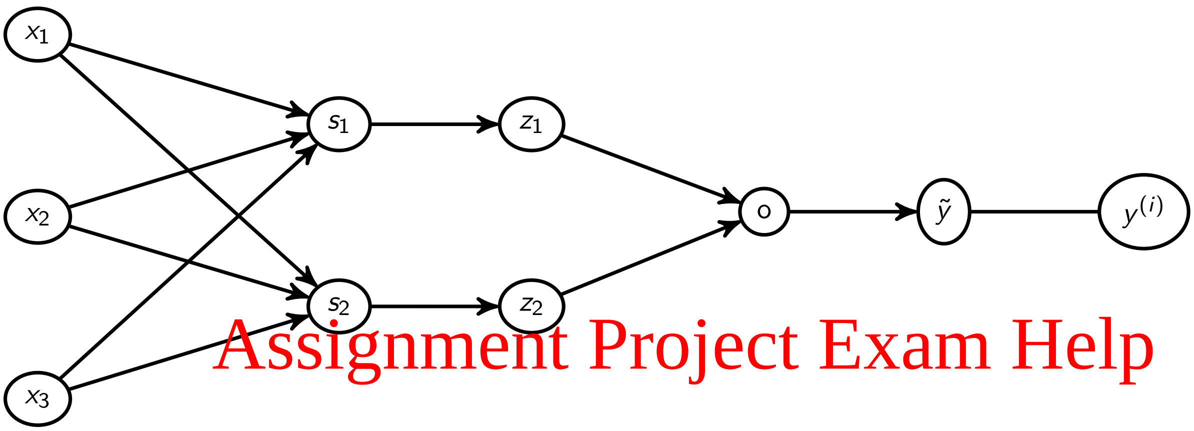
Derivative of the sigmoid activation function

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

$$= \frac{\partial \ell^{(i)}}{\partial z_k} z_k (1 - z_k) x_n$$

- If the negative log-likelihood $\ell^{(i)}$ does not depend much on z_k, then $\frac{\partial \ell^{(i)}}{\partial z_k} \approx 0$. In this case it doesn't matter how z_k is computed, and so $\frac{\partial \ell^{(i)}}{\partial \theta^{x \to z}_{n,k}} \approx 0$
- If $x_n = 0$, then it does not matter how we set the weights $\frac{\partial \ell^{(i)}}{\partial \theta^{x \to z}_{n,k}}$, so $\frac{\partial \ell^{(i)}}{\partial \theta^{x \to z}_{n,k}} = 0$

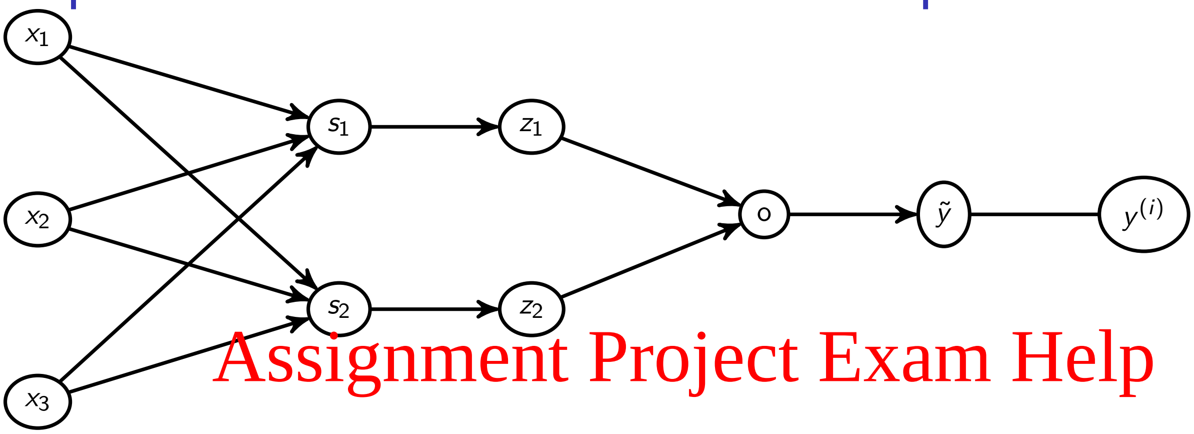
A simple neural network



$$
\mathbf{\Theta}^{(x \to s)} = \begin{array}{c} \\ s_1 \\ s_2 \end{array} \begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \begin{bmatrix} \theta_{11}^{(x\to s)} & \theta_{12}^{(x\to s)} & \theta_{13}^{(x\to s)} \\ \theta_{21}^{(x\to s)} & \theta_{22}^{(x\to s)} & \theta_{23}^{(x\to s)} \end{bmatrix} \end{array}
$$

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

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

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

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

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

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

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

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

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

Apply the chain rule:

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

$$\frac{\partial \mathcal{L}}{\partial \theta_{12}^{(z \to o)}} = \frac{\partial \mathcal{L}}{\partial o} \frac{\partial o}{\theta_{12}^{(z \to o)}} = (\tilde{y} - y) z_2$$

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

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

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

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

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

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

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

Vectorizing forward computation

$$\boldsymbol{s} = \boldsymbol{\Theta}^{(x \to s)} \boldsymbol{x} = \begin{bmatrix} \theta_{11}^{(x \to s)} & \theta_{12}^{(x \to s)} & \theta_{13}^{(x \to s)} \\ \theta_{21}^{(x \to s)} & \theta_{22}^{(x \to s)} & \theta_{23}^{(x \to s)} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$= \begin{bmatrix} \theta_{11}^{(x \to s)} x_1 + \theta_{12}^{(x \to s)} x_2 + \theta_{13}^{(x \to s)} x_3 \\ \theta_{21}^{(x \to s)} x_1 + \theta_{22}^{(x \to s)} x_2 + \theta_{23}^{(x \to s)} x_3 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$$

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

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

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

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 useful to **cache them**.
- We can only compute the derivatives of the parameters when we have all the necessary "inputs" demanded by the chain rule. So careful sequencing is necessary to take advantage of the cached derivatives.
- This combination of *sequencing*, *caching*, and *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 in the graph. For example, inputs to a feedforward neural network can be the feature vector extracted from a training (or test) instance
  - Backpropagation computes gradient of the loss with respect to all variables other than inputs, and propagates to parameters.
- Parameters include weights and bias. They do not have parents, and are initialized and then updated by gradient 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 (which can be genralized to SoftMax) function, the error signal at the output $\boldsymbol{o}$ is: $\boldsymbol{D_o} = [\tilde{y} - y^{(i)}]$
This error signal can then be used to compute the gradient with respect to $\boldsymbol{\Theta}^{(z \to o)}$

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

Backpropagation: Computing error signals

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

$$
\begin{aligned}
\boldsymbol{D_s} &= (\boldsymbol{\Theta}^{(z\rightarrow o)}\boldsymbol{D_o}) \odot \boldsymbol{F'} \\
&= \left(\begin{bmatrix} \frac{\partial o}{\partial z_1} & \frac{\partial o}{\partial z_2} \end{bmatrix} \times \begin{bmatrix} \frac{\partial \ell}{\partial o} \end{bmatrix} \right) \odot \begin{bmatrix} \frac{\partial z_1}{\partial s_1} & \frac{\partial z_2}{\partial s_2} \end{bmatrix} \\
&= \left(\begin{bmatrix} \theta_{11}^{(z\rightarrow o)} & \theta_{12}^{(z\rightarrow o)} \end{bmatrix} \times \begin{bmatrix} \tilde{y} - y \end{bmatrix} \right) \odot \begin{bmatrix} z_1(1-z_1) & z_2(1-z_2) \end{bmatrix} \\
&= \begin{bmatrix} (\tilde{y} - y)\theta_{11}^{(z\rightarrow o)} z_1(1-z_1) & (\tilde{y} - y)\theta_{12}^{(z\rightarrow o)} z_2(1-z_2) \end{bmatrix}
\end{aligned}
$$

Backpropagation: caching "error signals"

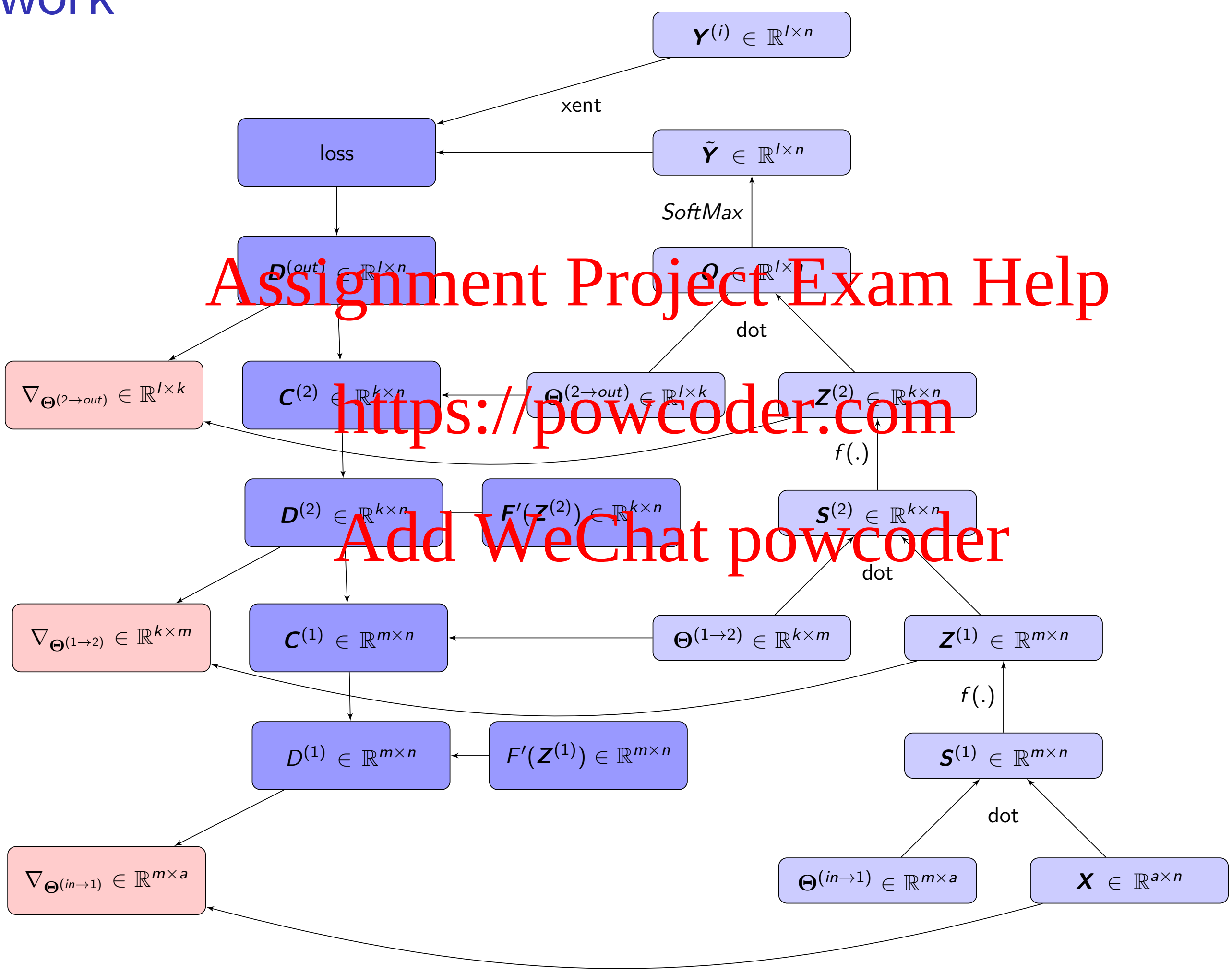
Using error signals $\boldsymbol{D_s}$, we can now compute the gradient on $\boldsymbol{\Theta}^{(x \to s)}$:

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

$$= \begin{bmatrix} (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1) \\ (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2) \end{bmatrix} \times \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$$

$$= \begin{bmatrix} (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_1 & (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_2 & (\tilde{y} - y)\theta_{11}^{(z \to o)} z_1(1 - z_1)x_3 \\ (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_1 & (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_2 & (\tilde{y} - y)\theta_{12}^{(z \to o)} z_2(1 - z_2)x_3 \end{bmatrix}$$

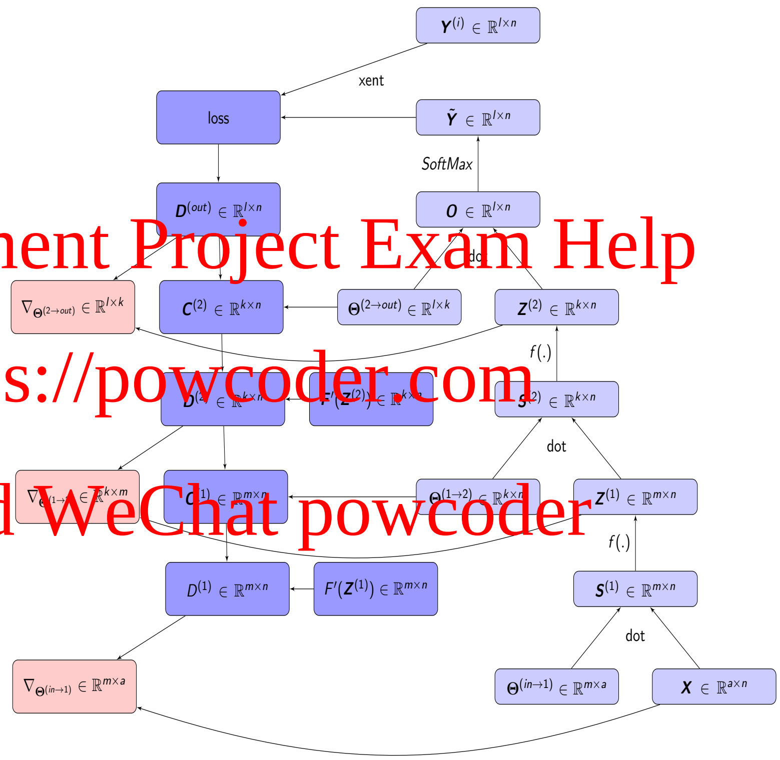
Computation graph of a three-layer feedforward neural network



Forward computation in matrix form

$$\boldsymbol{S}^{(1)} = \boldsymbol{\Theta}^{in\to 1}\boldsymbol{X}$$
$$\boldsymbol{Z}^{(1)} = f_1(\boldsymbol{S}^{(1)})$$
$$\boldsymbol{S}^{(2)} = \boldsymbol{\Theta}^{1\to 2}\boldsymbol{Z}^{(1)}$$
$$\boldsymbol{Z}^{(2)} = f_2(\boldsymbol{S}^{(2)})$$
$$\boldsymbol{O} = \boldsymbol{\Theta}^{2\to o}\boldsymbol{Z}^{(2)}$$
$$\tilde{\boldsymbol{y}}_{[j]} = \mathrm{SoftMax}(\boldsymbol{O}^\top{}_{[j]})$$

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

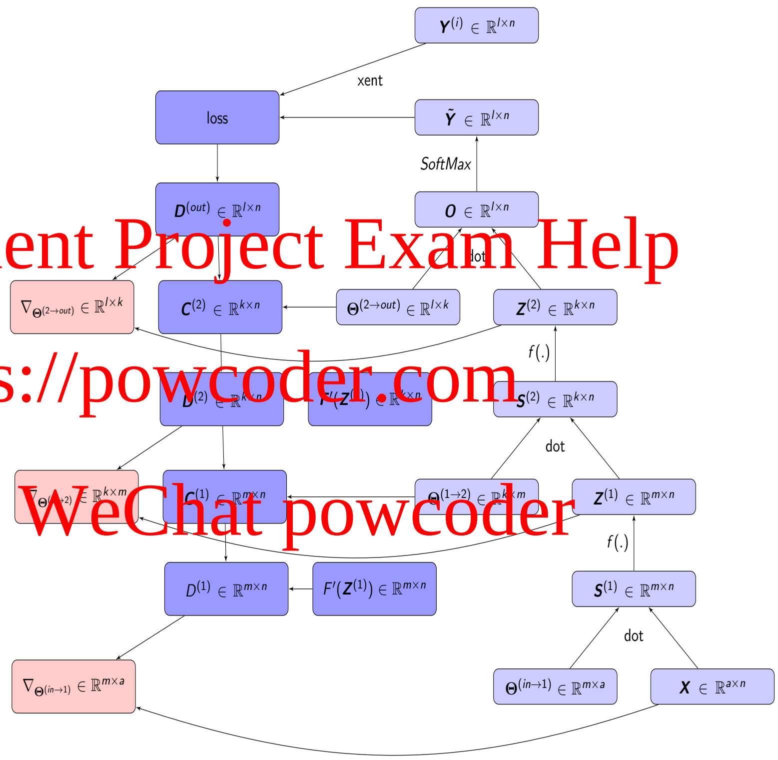
$$\boldsymbol{D}^{out} = \tilde{\boldsymbol{Y}} - \boldsymbol{Y}^{(i)}$$

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

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

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

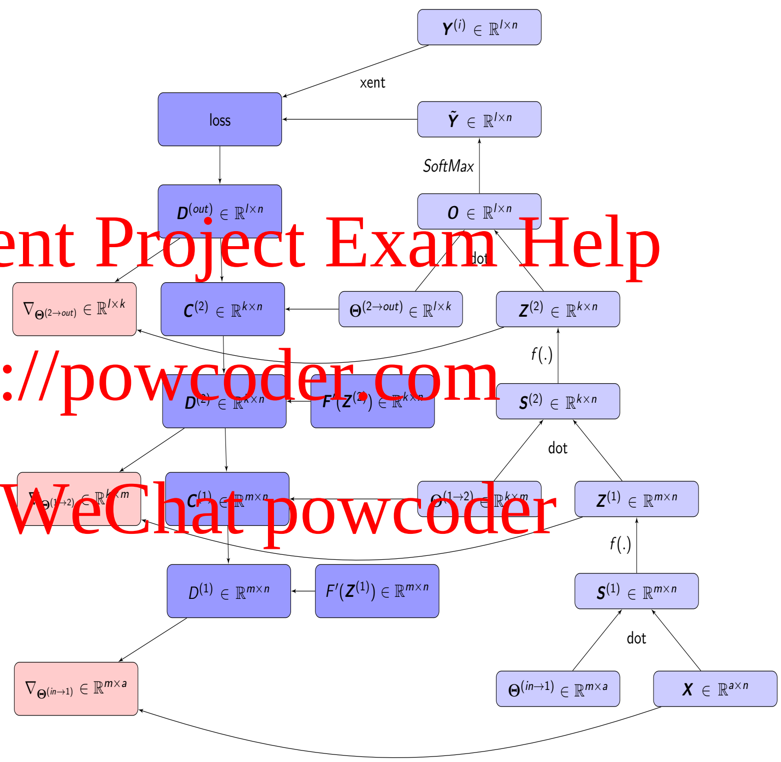
$$\boldsymbol{D}^{(1)} = \boldsymbol{F'}(\boldsymbol{Z}^{(1)}) \odot \boldsymbol{C}^{(1)}$$



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

Backpropagation: Compute the gradient with error signals

$$\nabla\boldsymbol{\Theta}^{2\rightarrow out} = \boldsymbol{D}^{(out)}(\boldsymbol{Z}^{(2)})^\top$$
$$\nabla\boldsymbol{\Theta}^{1\rightarrow 2} = \boldsymbol{D}^{(2)}(\boldsymbol{Z}^{(1)})^\top$$
$$\nabla\boldsymbol{\Theta}^{in\rightarrow 1} = \boldsymbol{D}^{(1)}(\boldsymbol{X})^\top$$



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

Notes in backpropagation

- Where does caching take place?
- Why is sequencing important?

- What computation patterns can you observe?

Pay attention to what you sum over

- In forward computation, you sum over all inputs (feature vector) for each output
- In backward computation, you sum over the gradient of all outputs for each input
- When you compute the gradient for the weights, you sum over the gradient for each data point (and average them)
- Make sure you line up the columns of the first matrix and the 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 automatic differentiation software libraries exist, e.g., Torch (PyTorch), MXNet, TensorFlow
- Using these libraries, users only need to specify the forward computation, and the gradient can be computed automatically. These libraries have accelerated research and development in deep learning considerably.
- Libraries that support dynamic computation graphs are better suited for many NLP problems.

Bells and whistles in neural net training

Tricks in training neural networks

There are various tricks that people use when training neural networks:

- Regularization: Adjusting the gradient
- Dropout: Adjusting the hidden units
- Optimization methods: Adjusting the learning rate
- Initialization: Using particular forms of initialization

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 many cases it is just referred to as L2 regularization.

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

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 gradient of a loss with L2 regularization

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

- Update the weights:

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

- "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 regularization loss

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

- Compute the gradient

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

- 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 value, $|\theta|$, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when $|\theta|$ is small, L1 regularization shrinks the weight much more than L2 regularization.
- The net result is that L1 regularization tends to concentrate 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 nodes to prevent over-reliance on a few features or hidden units, or **feature co-adaptation**, where some features are only useful when working together with a few other features. The ultimate goal is to avoid overfitting.

Dropout

- Dropout can be achieved using a mask:

$$
\begin{aligned}
\boldsymbol{z}^{(\mathbf{1})} &= g^1(\boldsymbol{\Theta}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{\mathbf{1}}) \\
\boldsymbol{m}^{\mathbf{1}} &\sim \mathit{Bernouli}(r^1) \\
\tilde{\boldsymbol{z}}^{(\mathbf{1})} &= \boldsymbol{m}^{\mathbf{1}} \odot \boldsymbol{z}^{(\mathbf{1})} \\
\boldsymbol{z}^{(\mathbf{2})} &= g^2(\boldsymbol{\Theta}^{(2)}\tilde{\boldsymbol{z}}^{(\mathbf{1})} + \boldsymbol{b}^{\mathbf{2}}) \\
\boldsymbol{m}^{\mathbf{2}} &\sim \mathit{Bernouli}(r^2) \\
\tilde{\boldsymbol{z}}^{(\mathbf{2})} &= \boldsymbol{m}^{\mathbf{2}} \odot \boldsymbol{z}^{(\mathbf{2})} \\
\boldsymbol{y} &= \boldsymbol{\Theta}^{(3)}\tilde{\boldsymbol{z}}^{(2)}
\end{aligned}
$$

where $\boldsymbol{m}^{\mathbf{1}}$ and $\boldsymbol{m}^{\mathbf{2}}$ 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 Square Prop (RMSProp)
- Adam

Momentum

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

$$\boldsymbol{V}_t = \gamma \boldsymbol{V}_{t-1} + \eta \nabla_{\boldsymbol{\Theta}}(L)$$

$$\boldsymbol{\Theta} = \boldsymbol{\Theta} - \boldsymbol{V}_t$$

The momentum term increases for dimensions whose gradient 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:

$$\boldsymbol{V}_{\nabla_\Theta} = \boldsymbol{V}_{\nabla_\Theta} + \boldsymbol{\nabla}^2_{\boldsymbol{\Theta}}$$

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

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

Root Mean Square Prop (RMSProp)

Weight update for RMSprop at each time step t:

$$\boldsymbol{S}_{\nabla_{\Theta}} = \beta \boldsymbol{S}_{\nabla_{\Theta}} + (1-\beta)\boldsymbol{\nabla}^2_{\boldsymbol{\Theta}}$$

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

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

Adaptive Moment Estimation (Adam)

Weight update at time step t for Adam:

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

$$\boldsymbol{S}_{\nabla_\Theta} = \beta_2 \boldsymbol{S}_{\nabla_\Theta} + (1 - \beta_2)\boldsymbol{\nabla_\Theta^2}$$

$$\boldsymbol{V}_{\nabla_\Theta}^{corrected} = \frac{\boldsymbol{V}_{\nabla_\Theta}}{\beta_1^t}$$

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

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

Adam combines Momentum and RMSProp

Define a neural net

```
from torch import nn
class Net(nn.Module):
    '''subclass from nn.Module is important to insp
    def __init__(self, in_dim=25, out_dim=3, batch_
        super(Net, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.linear = nn.Linear(self.in_dim, self.o
        self.softmax = nn.Softmax(dim=1) #softmax o

    def forward(self, input_matrix):
        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.optim as optim
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_data, batch_size):
        optimizer.zero_grad() #zero out the gradient.
        vectorized = vectorize_batch(batch, feat_index, label
        feat_vec = map(itemgetter(0), vectorized)
        label_vec = map(itemgetter(1), vectorized)
        feat_list = list(feat_vec)
        label_list = list(label_vec)
        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 neural networks

Input to feedforward neural networks

- Assuming a bag-of-words model, when the input $\boldsymbol{x}$ is the count of each word (feature) x_i .
- To compute the hidden unit z_k:

$$z_k = \sum_{j=1}^{V} \theta_{j,k}^{x \to z} x_j$$

- The connections from word j to each of the hidden units z_k form a vector $\theta_j^{(x \to z)}$ is sometimes described as the embedding of word j.

- If there is a lot of training data, word embeddings can be learned within the same network as the classification task.

- 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 classification instance can be thought of as the sum of indicator feature vectors 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 representation over a vocabulary 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

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

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 of one another. The feature “word is ‘dog’ ” is as dissimilar to “word is ‘thinking’ ” as it is to “word is ‘cat”’
  - Features for one classifying instance can only combined by 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 $f_1, \cdots, f_k$ that are relevant for predicting the output class
- For each feature f_i of interest, retrieve the corresponding vector v_i.
- Combine the vectors (either by concatenation, summation, or a combination of both) into an input vector x.
  - 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 representations are typically pre-computed or 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 when training a neural network amounts to dedicating the first layer of the network to learning a dense embedding vector [for each feature] based on training data.

- With task-specific word embedding, the training set is 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 Distributional Semantic Models (DSM) or Vector Semantic Models (VSM)
- Predictive methods, originating from the neural network community, aimed at producing Distributed Representations for words, commonly known as word embeddings
  - Distributed word representations were initially a by-product of neural language models and later became a separate task on 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 it as a slogan: "You shall know a word by the company it keeps." (J. R. Firth, 1957)
- A long history of using word-context matrices to represent word meaning where each row is a word and each column represents a context word it can occur with in a corpus
- Each word is represented as a sparse vector in a 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 syntactic parsing
- Define the "context" for a target term (words or phrases). The context can be a window centered on the target term, terms that are syntactically related to the target term (subject-of, object-of, etc.).
- Compute a term-context matrix where each row corresponds to a term and each column corresponds to a context term for 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-context matrix with association strength metrics such as Positive Pointwise Mutual Information (PPMI) to correct frequency bias

$$PPMI(x, y) = max(\log \frac{p(x, y)}{p(x)p(y)}, 0)$$

- Its dimensionality can also be reduced by matrix factorization techniques such as *singular value decomposition* (SVD)

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

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

- 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 embeddings from large naturally occurring text, using various language model objectives.
- Decide on the context window
- Define the objective function that is used to predict the context words based on the target word or predict the target word based on context
- Train the neural network
- The resulting weight matrix will serve as the vector 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) and Skip-gram.
- It popularized the use of distributed representations as input to neural networks in natural language processing, and inspired many follow-on works, e.g., GLOVE, ELMO, BERT, XLNet)
- It has it roots in language modeling (the use of window-based context to predict the target word), but gives up the goal of getting good language models and focus instead on getting good word embeddings.

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

- Input $\boldsymbol{x} \in \mathbb{R}^V$ is a one-hot vector where $x_k = 1$ 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 $\boldsymbol{v}_w$ of the associated word of the input layer.

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

- $\boldsymbol{\Theta'} \in \mathbb{R}^{V \times N}$ is the matrix from the hidden layer to the output layer and $\boldsymbol{u}_{w_j}$ is the j-th row of $\boldsymbol{\Theta'}$. A "similarity" score o_j for each target word w_j and context word w_i can be computed 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 "retrieves" $\boldsymbol{v}_{w_i}$:

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

$$\begin{bmatrix} 0.1 & 0.3 & 0.5 & 0.4 & 0.6 & 0.1 & 0.3 & 0.5 & 0.4 & 0.6 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \\ 0.2 & 0.5 & 0.8 & 0.7 & 0.9 & 0.4 & 0.8 & 0.2 & 0.5 & 0.1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix}$$

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

$$
\boldsymbol{o} = \boldsymbol{\Theta}'\boldsymbol{z} = \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} \times \begin{bmatrix} 0.5 \\ 0.8 \\ 0.8 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.91 \\ 0.97 \\ 0.82 \\ 1.18 \\ 0.31 \\ 1.06 \\ 0.39 \\ 0.95 \\ 0.63 \end{bmatrix}
$$

Each row of $\boldsymbol{\Theta}'$ correspond to vector for a target word w_j.

Taking the softmax over the output

$$\boldsymbol{y} = softmax(\boldsymbol{o}) = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ 0.11262222 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}$$

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 input context word and the vector of the output target word.

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

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 cross-entropy loss

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

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

- Given y_j is the output of a softmax function, the gradient 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$\rightarrow$output weights

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

Updating input→hidden weights

- Compute the error at the hidden layer

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

- Since

$$z_i = \sum_{k=1}^{V} \theta_{ik} x_k$$

The derivative of $\mathcal{L}$ on the input→ hidden weights:

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

- 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

$$D_o = Y - T = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ 0.11262222 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.11039215 \\ 0.10606362 \\ -0.88737778 \\ 0.09693485 \\ 0.13893957 \\ 0.05820895 \\ 0.12322834 \\ 0.063057 \\ 0.11039215 \\ 0.08016116 \end{bmatrix}$$

Computing the errors at the hidden layer

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

$$\begin{bmatrix} 0.110 & 0.106 & -0.887 & 0.097 & 0.139 & 0.058 & 0.123 & 0.063 & 0.110 & 0.080 \end{bmatrix}$$

$$\times \begin{bmatrix} 0.3 & 0.4 & 0.6 \\ 0.7 & 0.1 & 0.6 \\ 0.5 & 0.2 & 0.7 \\ 0.2 & 0.6 & 0.3 \\ 0.6 & 0.5 & 0.6 \\ 0.3 & 0.1 & 0.1 \\ 0.2 & 0.4 & 0.8 \\ 0.3 & 0.2 & 0.1 \\ 0.3 & 0.4 & 0.6 \\ 0.3 & 0.5 & 0.1 \end{bmatrix} = \begin{bmatrix} -0.115 & 0.157 & -0.194 \end{bmatrix}$$

Computing the updates to $\mathbf{\Theta}'$

$$\nabla_{\mathbf{\Theta}'} = \boldsymbol{D_o z}^\top =$$

$$\begin{bmatrix} 0.110 \\ 0.106 \\ -0.887 \\ 0.097 \\ 0.139 \\ 0.058 \\ 0.123 \\ 0.063 \\ 0.110 \\ 0.080 \end{bmatrix} \times \begin{bmatrix} 0.5 & 0.8 & 0.8 \end{bmatrix} = \begin{bmatrix} 0.055 & 0.088 & 0.088 \\ 0.053 & 0.085 & 0.085 \\ -0.443 & -0.710 & -0.710 \\ 0.048 & 0.078 & 0.0778 \\ 0.069 & 0.111 & 0.111 \\ 0.029 & 0.047 & 0.0467 \\ 0.062 & 0.099 & 0.099 \\ 0.032 & 0.050 & 0.050 \\ 0.055 & 0.088 & 0.088 \\ 0.040 & 0.064 & 0.064 \end{bmatrix}$$

Computing the update to $\boldsymbol{\Theta}$

$$\nabla_{\boldsymbol{\Theta}} = \boldsymbol{x}\boldsymbol{D} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} -0.11538456 & 0.15687943 & -0.19388611 \end{bmatrix}$$

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

CBOW for multiple context words

$$\begin{aligned} \boldsymbol{z} &= \frac{1}{M}\boldsymbol{\Theta}(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \cdots \boldsymbol{x}_M) \\ &= \frac{1}{M}(\boldsymbol{v}_{w_1} + \boldsymbol{v}_{w_2} + \cdots + \boldsymbol{v}_{w_M}) \end{aligned}$$

where M is the number of words in the context, $w_1, w_2, \cdots, w_M$ are the words in the context and $\boldsymbol{v}_w$ is an input vector. The loss function is

$$\begin{aligned} E &= -\log p(w_j | w_1, w_2, \cdots, w_M) \\ &= -o_{j^\star} + \log \sum_{j'=1}^{V} \exp(o_{j'}) \\ &= -\boldsymbol{u}_{w_j}^\top \boldsymbol{z} + \log \sum_{j'=1}^{V} \exp(\boldsymbol{u}_{w_{j'}}^\top \boldsymbol{z}) \end{aligned}$$

Computing the hidden layer for multiple context words

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

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

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

Skip-gram: model

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{exp(o_{c,j})}{\sum_{j'=1}^{V} exp(o_{j'})}$$

where $w_{c,j}$ is the j-th word on the c-th panel of the output layer, $w_{O,c}$ is the actual c-th word in the output context words; w_I is the only input word, $y_{c,j}$ is the output of the j-th unit on the c-th panel of the output layer; $o_{c,j}$ is the net input of the 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

$$
\begin{aligned}
\mathcal{L} &= -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C} | w_I) \\
&= -\log \prod_{c=1}^{C} \frac{exp(o_{c,j_c^\star})}{\sum_{j'=1}^{V} exp(o_{j'})} \\
&= -\sum_{c=1}^{C} o_{c,j_c^\star} + C \times \log \sum_{j'=1}^{V} exp(o_{j'})
\end{aligned}
$$

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 $\mathcal{L}$ with regard to the net input of every unit on every panel of the output layer, $o_{c,j}$, 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 a V-dimensional vector $E = E_1, \cdots, E_V$ as the sum of the prediction errors of the context word: $E_j = \sum_{c=1}^{C} e_{c,j}$

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

- Updating the hidden$\rightarrow$output weight matrix:

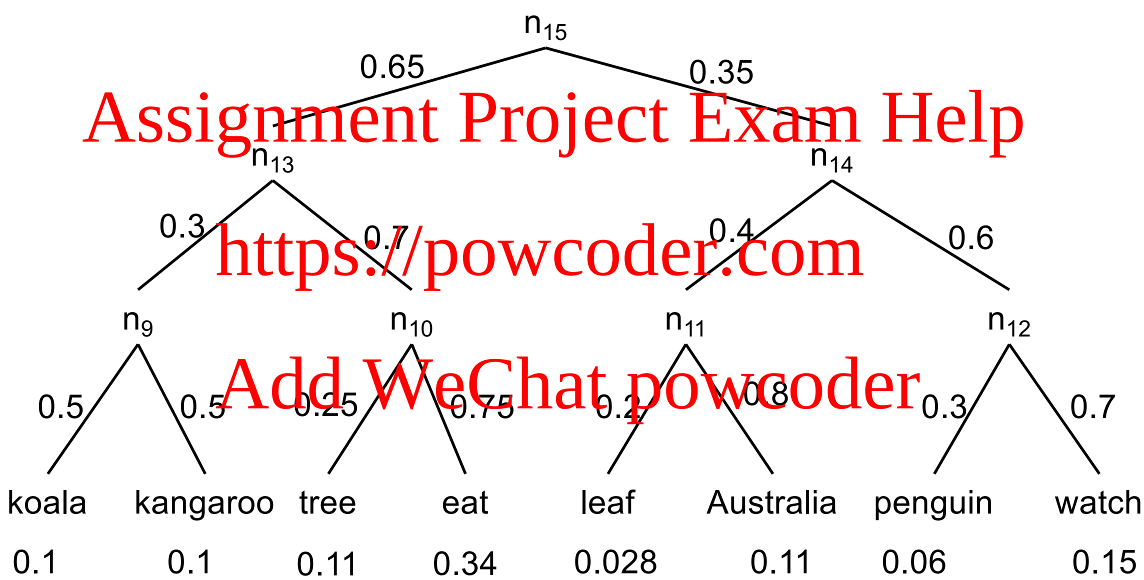
$$\theta'_{ji} = \theta'_{ji} - \eta \cdot E_j \cdot z_i$$

- No change in how the input$\rightarrow$hidden weights are updated.

Optimizing computational efficiency

- Computing softmax at the output layer is expensive. It involves iterative over the entire vocabulary
- Two methods for optimizing computational efficiency
  - **Hierarchical softmax**: an alternative way to compute the probability of a word that reduces the computation complexity from $|V|$ to $\log |V|$.
  - **Negative sampling**: Instead of updating the weights for 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

$$P(\text{“}Kangaroo\text{''}|z) = P_{n_{15}}(Left|\mathbf{z}) \times P_{n_{13}}(Left|\mathbf{z}) \times P_{n_9}(Right|\mathbf{z})$$

$$P_n(Right|\mathbf{z}) = 1 - P_n(Left|\mathbf{z})$$

$$P_n(Left|\mathbf{z}) = \sigma(\boldsymbol{\gamma}_n^\top \mathbf{z})$$

where $\boldsymbol{\gamma}_n$ 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 Huffman trees, each of which is a single leaf node. Put the n trees onto a priority queue organized by weight (frequency).
- Remove the first two trees (the ones with lowest weight). Join these two trees to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two children trees. Put this new tree into the priority 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 $P(D = 1|w, c)$ be the probability of the pair of words came from the training corpus, and $P(D = 0|w, c)$ be the probability that the pair did not come from the corpus.
- This probability can be modeled as a sigmoid function:

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

New learning objective for negative sampling

- We need a new objective for negative sampling, which is to minimize the following loss function:

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

  where D is a set of correct context - target word pairs and D' is a set of incorrect context - target word pairs.
- Note that we use the sum for positive samples as well as negative samples. In the skip-gram algorithm, there will be multiple positive context words in the output. In the CBOW 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 weights

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

- When updating the output weights, only the weight vectors for words in the positive sample and negative sample need to be updated:

Updates to the input→hidden weights

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

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \sum_{w_j \in D \cup D'} (\sigma(o_{w_j}) - t_{w_j}) \boldsymbol{u}_{w_j}$$

- In the CBOW algorithm, the weights for all input context words will be updated. In the Skip-gram algorithm, the weight 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 pick a word from a corpus, the probability of any given word w_i getting picked is:

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

  More frequent words will be more likely to be picked and this may not be ideal.

- Adjust the formula to give the less frequent words a bit more 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

$$sim_{cos} = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{||\boldsymbol{u}||_2 ||\boldsymbol{v}||_2}$$

- How accurately can they be used to compute word similarity can also be used to evaluate word embeddings

- They can also be used to compute the similarity of short documents

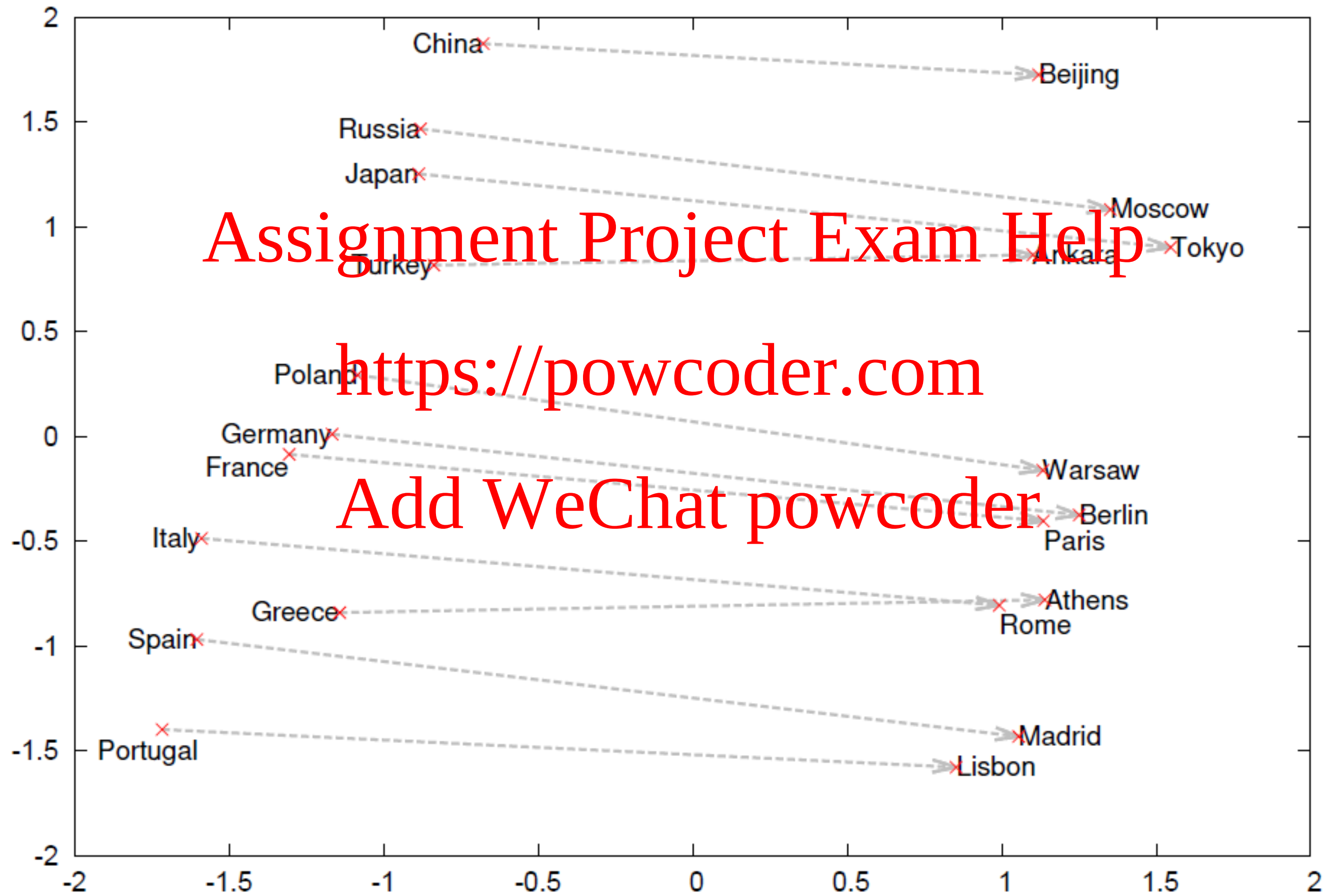
$$sim_{doc}(D_1, D_2) = \sum_{i=1}^{m} \sum_{j=1}^{n} cos(\boldsymbol{w_i^1}, \boldsymbol{w_j^2})$$

Use of embeddings: word analogy

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

$$
\begin{aligned}
analogy(m : w \rightarrow k :?) &= \operatorname*{argmax}_{v \in V \setminus m,k,w} cos(\boldsymbol{v}, \boldsymbol{k} - \boldsymbol{m} + \boldsymbol{w}) \\
analogy(m : w \rightarrow k :?) &= \operatorname*{argmax}_{v \in V \setminus m,k,w} cos(\boldsymbol{v}, \boldsymbol{k}) - cos(\boldsymbol{v}, \boldsymbol{m}) \\
&+ cos(\boldsymbol{v}, \boldsymbol{w}) \\
analogy(m : w \rightarrow k :?) &= \operatorname*{argmax}_{v \in V \setminus m,k,w} \frac{cos(\boldsymbol{v}, \boldsymbol{k}) cos(\boldsymbol{v}, \boldsymbol{w})}{cos(\boldsymbol{v}, \boldsymbol{m}) + \epsilon}
\end{aligned}
$$

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 as input to predict the outcome of tasks that have real-world applications
- Many follow-on work in develop more effective word embeddings, e.g., GLOVE
  - word2vec: http://vectors.nlpl.eu/repository
  - fasttext: https://fasttext.cc/docs/en/english-vectors.html
  - GLOVE: https://nlp.stanford.edu/projects/glove
- 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

```
In [39]: from torch import nn
         embedding = nn.Embedding(10, 3)
         print(embedding)
         input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
         embedding(input)

         Embedding(10, 3)

Out[39]: tensor([[[-0.9538,  0.3385, -1.6404],
                  [ 1.7206,  1.4395,  0.2744],
                  [-2.9429,  0.9432, -0.4569],
                  [-0.6187, -0.5479,  0.2746]],

                 [[-2.9429,  0.9432, -0.4569],
                  [ 1.2738,  1.1245,  0.6983],
                  [ 1.7206,  1.4395,  0.2744],
                  [ 0.6431, -1.2324, -1.3246]]], grad_fn=<EmbeddingBackward>)
```

```
In [38]: weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
         embedding = nn.Embedding.from_pretrained(weight)
         input = torch.LongTensor([0,1,1])
         embedding(input)

Out[38]: tensor([[1.0000, 2.3000, 3.0000],
                 [4.0000, 5.1000, 6.3000],
                 [4.0000, 5.1000, 6.3000]])
```

Commonly used neural architectures

- One important aspect of neural network modeling is to figure out the right representation for a problem
- Commonly used architectures
  - Variants of the Recurrent Neural Networks (RNN), which have been instrumental in improving many states of the art in NLP
  - Convolutional Networks (CNN), which have been very effective in image processing and some NLP problems (e.g., sentence classification)

Convolutional Networks for text classification

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 structure with a pooling function, capturing the local aspects that are most informative of the prediction task at hand.
- A convolutional network is not fully connected as a feedforward network is.
- It has been tremendously successful in image procession (or 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

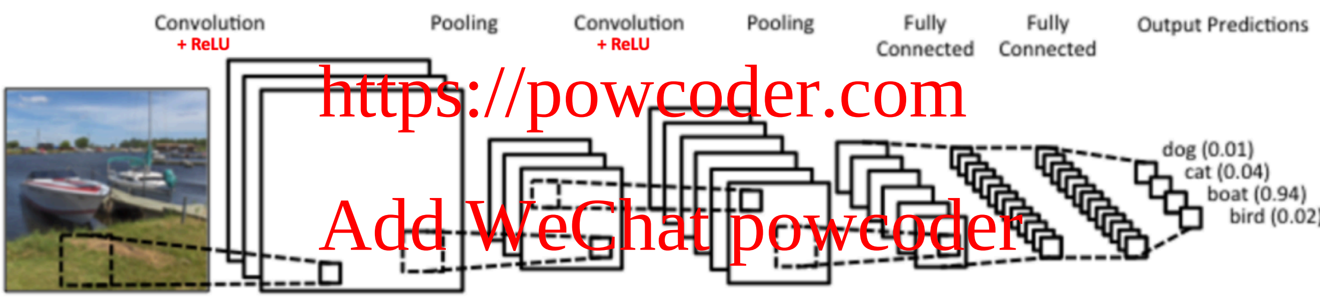


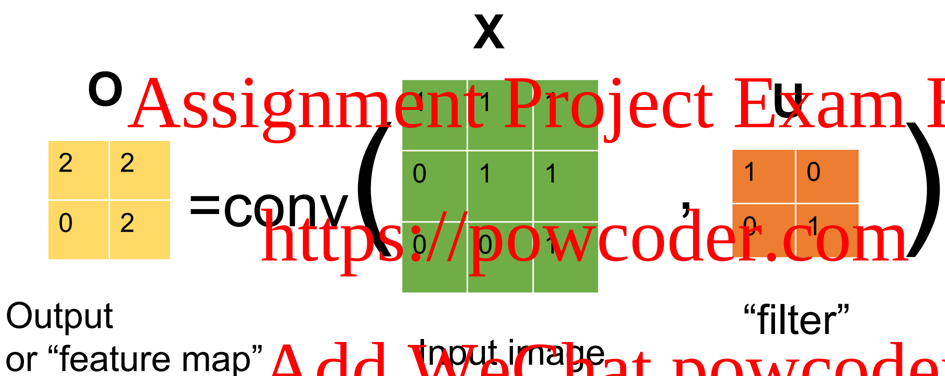
Image pixels

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	12	0	11	39	137	37	0	152	147	84	0	0	0
0	0	1	0	0	0	41	160	250	255	235	162	255	238	206	11	13	0
0	0	0	16	9	9	[illegible]	251	45	21	184	159	[illegible]	255	233	40	0	0
[illegible]	0	0	0	0	0	[illegible]	[illegible]	[illegible]	[illegible]	0	11	[illegible]	[illegible]	[illegible]	187	0	0
0	0	3	0	4	15	236	216	0	0	38	109	247	240	169	0	11	0
1	0	2	0	0	0	253	253	23	62	224	241	255	164	0	5	0	0
6	[illegible]	0	[illegible]	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0	[illegible]	[illegible]	0
0	2	1	4	0	21	255	253	251	255	172	31	8	0	1	0	0	0
0	0	4	0	163	225	251	255	229	120	0	0	0	0	0	11	0	0
0	0	21	162	255	[illegible]	254	255	126	6	0	10	14	6	0	0	9	0
3	79	242	255	141	66	255	245	189	7	8	0	0	5	0	0	0	0
26	221	237	98	0	67	251	255	144	0	8	0	0	7	0	0	11	0
125	255	141	0	87	244	255	208	3	0	0	13	0	1	0	1	0	0
145	248	228	116	235	255	141	34	0	11	0	1	0	0	0	1	3	0
85	237	253	246	255	210	21	1	0	1	0	0	6	2	4	0	0	0
6	23	112	157	114	32	0	0	0	0	2	0	8	0	7	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Four operations in a convolutional network

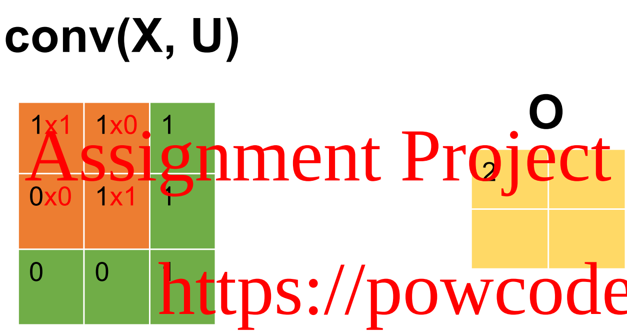
- Convolution
- Non-linear activation (ReLU)
- Pooling or subsampling (Max)
- Classification with a fully connected layer

Image convolution



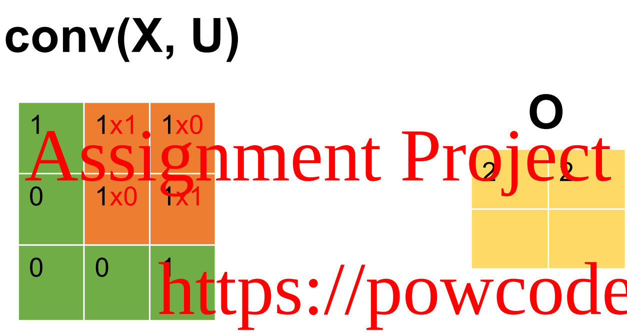
Kernel size = (2,2), strides = 1

Image convolution



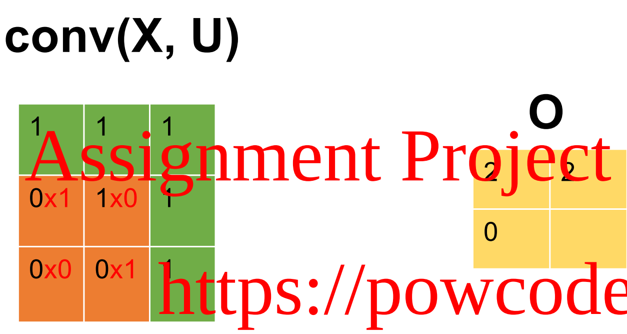
Kernel size = (2,2), strides = 1

Image convolution



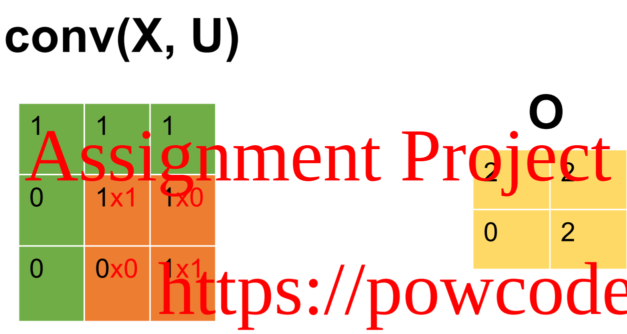
Kernel size = (2,2), strides = 1

Image convolution



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

Image convolution



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

Forward computation

$$
\begin{aligned}
o_{11} &= x_{11}u_{11} + x_{12}u_{12} + x_{21}u_{21} + x_{22}u_{22} \\
o_{12} &= x_{12}u_{11} + x_{13}u_{12} + x_{22}u_{21} + x_{23}u_{22} \\
o_{21} &= x_{21}u_{11} + x_{22}u_{12} + x_{31}u_{21} + x_{32}u_{22} \\
o_{22} &= x_{22}u_{11} + x_{23}u_{12} + x_{32}u_{21} + x_{33}u_{22}
\end{aligned}
$$

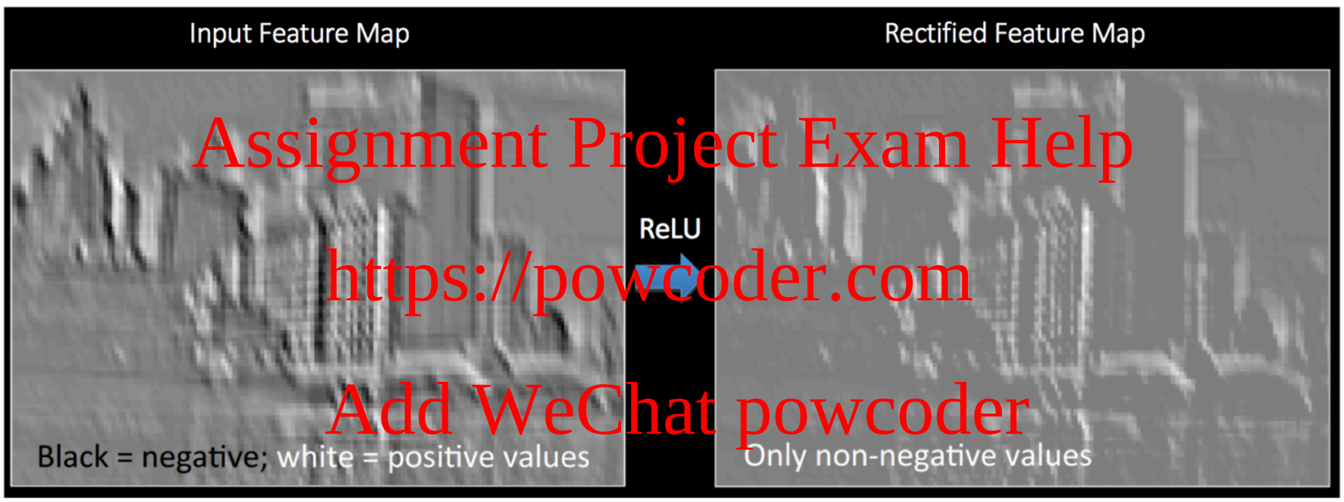
ReLU

- Nonlinear transformation with ReLU

$$Output = ReLU(input) = \max(0, input)$$

- As we know, ReLU is an element-wise transformation that does not change the dimension of the feature map
- ReLU replaces all negative pixel values in the featuremap with 0

Image ReLU


Input Feature Map
Rectified Feature Map
ReLU
Black = negative; white = positive values
Only non-negative values

ReLU activation and Max pooling

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

$$\begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix} = ReLU\left(\begin{bmatrix} 2 & 2 \\ 0 & 2 \end{bmatrix}\right)$$

- Max pooling does change the dimension of the input. Need to specify the pool size and strides.

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

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

Training a CNN

- Loss functions: Cross-entropy loss, squared error loss
- What are the parameters of a CNN?
  - The filters (kernels), weight matricies for the feedforward network on top of the convolution and pooling layers, biases
- Computing the gradient for the convolution layers is different from a feedforward neural network...

Computing the gradient on $\boldsymbol{U}$

$$
\begin{aligned}
\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_{21}}\frac{\partial o_{21}}{\partial u_{12}} + \frac{\partial E}{\partial o_{22}}\frac{\partial o_{22}}{\partial u_{12}} \\
&= \frac{\partial E}{\partial o_{11}}x_{12} + \frac{\partial E}{\partial o_{12}}x_{13} + \frac{\partial E}{\partial o_{21}}x_{22} + \frac{\partial E}{\partial o_{22}}x_{23} \\
\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_{21}}\frac{\partial o_{21}}{\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}
\end{aligned}
$$

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 vectorized as a reverse convolution:

$$\begin{bmatrix} \frac{\partial E}{\partial u_{11}} & \frac{\partial E}{\partial u_{12}} \\ \frac{\partial E}{\partial u_{21}} & \frac{\partial E}{\partial u_{22}} \end{bmatrix} = conv \left(\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial E}{\partial o_{11}} & \frac{\partial E}{\partial o_{12}} \\ \frac{\partial E}{\partial o_{21}} & \frac{\partial E}{\partial o_{22}} \end{bmatrix} \right)$$

Computing the gradient on $\boldsymbol{X}$ (if this is not the input layer)

$$
\begin{aligned}
\frac{\partial E}{\partial x_{11}} &= \frac{\partial E}{\partial o_{11}} u_{11} + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} 0 \\
\frac{\partial E}{\partial x_{12}} &= \frac{\partial E}{\partial o_{11}} u_{12} + \frac{\partial E}{\partial o_{12}} u_{11} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} 0 \\
\frac{\partial E}{\partial x_{13}} &= \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} u_{12} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} 0 \\
\frac{\partial E}{\partial x_{21}} &= \frac{\partial E}{\partial o_{11}} u_{21} + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} u_{11} + \frac{\partial E}{\partial o_{22}} 0 \\
\frac{\partial E}{\partial x_{22}} &= \frac{\partial E}{\partial o_{11}} u_{22} + \frac{\partial E}{\partial o_{12}} u_{21} + \frac{\partial E}{\partial o_{21}} u_{12} + \frac{\partial E}{\partial o_{22}} u_{11} \\
\frac{\partial E}{\partial x_{23}} &= \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} u_{22} + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} u_{12} \\
\frac{\partial E}{\partial x_{31}} &= \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} u21 + \frac{\partial E}{\partial o_{22}} 0 \\
\frac{\partial E}{\partial x_{32}} &= \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} u_{22} + \frac{\partial E}{\partial o_{22}} u_{21} \\
\frac{\partial E}{\partial x_{12}} &= \frac{\partial E}{\partial o_{11}} 0 + \frac{\partial E}{\partial o_{12}} 0 + \frac{\partial E}{\partial o_{21}} 0 + \frac{\partial E}{\partial o_{22}} u_{22}
\end{aligned}
$$

Full convolution

$$\begin{bmatrix} \frac{\partial E}{\partial x_{11}} & \frac{\partial E}{\partial x_{12}} & \frac{\partial E}{\partial x_{13}} \\ \frac{\partial E}{\partial x_{21}} & \frac{\partial E}{\partial x_{22}} & \frac{\partial E}{\partial u_{23}} \\ \frac{\partial E}{\partial x_{31}} & \frac{\partial E}{\partial x_{32}} & \frac{\partial E}{\partial u_{33}} \end{bmatrix} = full\_conv \left(\begin{bmatrix} \frac{\partial E}{\partial o_{11}} & \frac{\partial E}{\partial o_{12}} \\ \frac{\partial E}{\partial o_{21}} & \frac{\partial E}{\partial o_{22}} \end{bmatrix}, \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \right)$$

Gradient on $\boldsymbol{X}$ if it is not the inputs

Sample code of 2D convolution with Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```


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 simple and work well in some text classification tasks, they don't account for cases where multiple words combine to create meaning, such as "not interesting".
- The analogy with image processing is if the pixels are treated as separate features. (The analogy might be going too far).

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 Word2Vec or GLOVE[1]) and the input sentence:

$$\boldsymbol{X}^{(0)} = \boldsymbol{\Theta}^{(x \to z)}[\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \cdots, \boldsymbol{e}_{w_M}]$$

$$\boldsymbol{X}^{(0)} \in \mathbb{R}^{K_e \times M}$$

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 w_m is represented as a one-hot vector $\boldsymbol{e}_{w_m}$, with dimension V. The complete document can be represented by the horizontal concatenation of these one-hot vectors: $\boldsymbol{W} = [\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \cdots, \boldsymbol{e}_{w_m}] \in R^{V \times M}$.
- 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 have to cover the full width of the image.
- To merge adjacent words, we *convolve* $\boldsymbol{X}^{(0)}$ by sliding a set of filters across it:

$$\boldsymbol{X}^{(1)} = f(\boldsymbol{b} + \boldsymbol{C} * \boldsymbol{X}^{(0)})$$

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 (the mth word in the sequence), 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 at m and take its sum: $\boldsymbol{C}^{(k)} \odot \boldsymbol{X}^{(0)}_{m:m+h-1}$
- The value of the mth position with the kth filter can be computed as:

$$x_m^{(1)} = f\left(b_k + \sum_{k'=1}^{K_e} \sum_{n=1}^{h} c_{k',n}^{(k)} \times x_{k',m+n-1}^{(0)}\right)$$

- 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

$$\text{Conv}\left(\overset{\mathbf{X}^{(0)}}{\underset{\text{Input text sequence}}{\begin{bmatrix} 0.4 & 0.7 & 0.8 & 0.3 & 2.3 & 3.2 & 0.7 \\ 0.5 & 1.3 & 0.2 & 1.5 & 0.8 & 0.6 & 1.8 \\ 1.2 & 2.2 & 1.1 & 4.3 & 0.5 & 0.3 & 3.4 \end{bmatrix}}} , \overset{\mathbf{U}}{\underset{\text{“filter”}}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}}} \right)$$

$$= \underset{\mathbf{X}^{(1)}}{\begin{bmatrix} 2.9 & 3.1 & 3.4 & ? & ? & ? \end{bmatrix}}$$

Padding

- To deal with the beginning and end of the input, the base matrix is often padded with $h-1$ column vectors of zeros at the beginning and end. this is called **wide convolution**
- If no padding is applied, then the output of each convolution layer will be $h-1$ units smaller 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 document as a matrix $\boldsymbol{X}^{(D)} \in \mathbb{R}^{K_L \times M}$.
- It is very likely that the documents will be of different lengths, so we need to turn them into matricies of the same length before feeding them to a feedward network to perform 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 matrics for the feedforward network and the filters $\mathbf{C}^{(k)}$ of the CNN, as well as the biases.
- The parameters can be updated with backpropagation, which may involve computing the gradient for the max pooling 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

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

- Average pooling

$$z_k = \frac{1}{M}\sum_{m=1}^{M} x_{k,m}^{(D)}$$

A graphic representation of a CNN

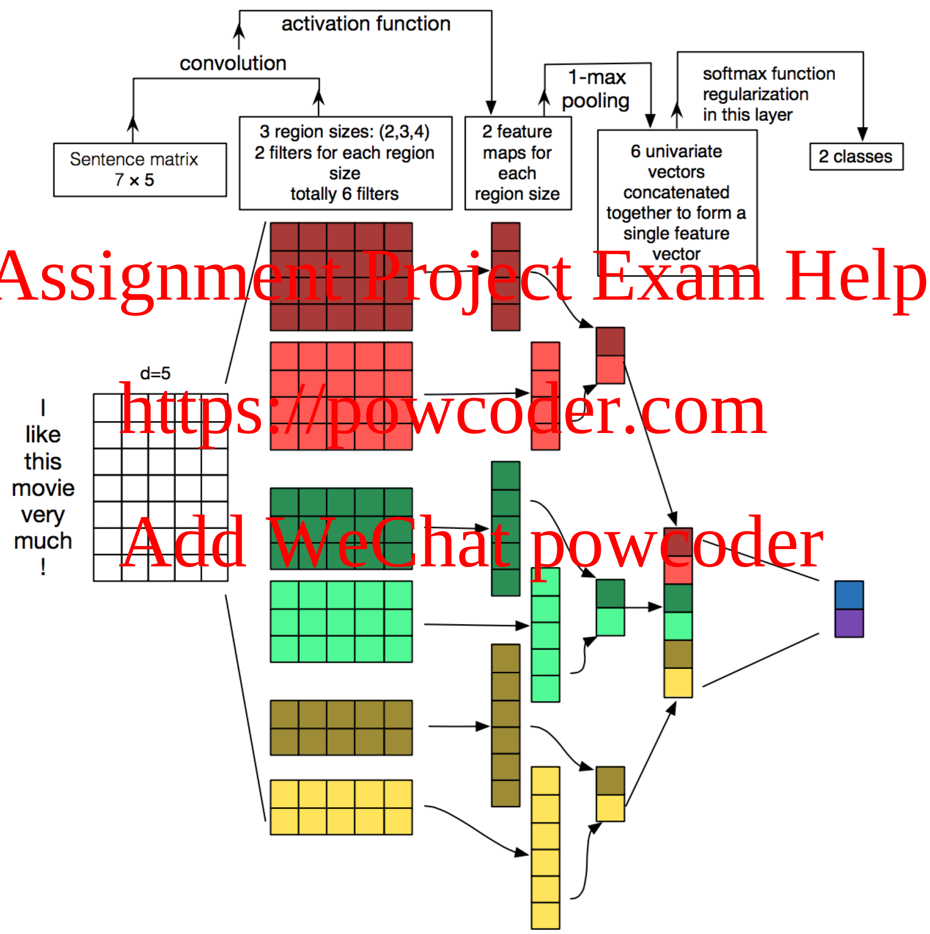


Figure 1: Caption

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,
                    output_dim=embedding_dim,
                    input_length=maxlen))
model.add(Conv1D(num_filters, kernel_size, \\
          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 non-linear models we have discussed so far, we assume we have labeled training data where we can perform supervised learning:
  - a **training set** where you get observations $\boldsymbol{x}$ and labels y;
  - a **test set** where you only get observations $\boldsymbol{x}$
- A summary of the supervised learning models we have discussed so far:
  - Linear models: Naïve Bayes, Logistic Regression, Perceptron, Support Vector Machines
  - Non-linear models: feed-forward networks, Convolutional 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-dependence between the labels (y) of the classification instances:
  - Word sense disambiguation
  - Sentiment and opinion analysis
  - Genre classification
  - Others
- NLP problems that cannot be formulated as simple text 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 do **unsupervised learning** (e.g., K-Means clustering, variants of the EM algorithm)
- When there is a small amount of labeled data, we might want 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$(1, K)$ ▷ Intializing class membership

  repeat

    for $k \in 1 \cdots K$ **do** ▷ recompute cluster centers

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

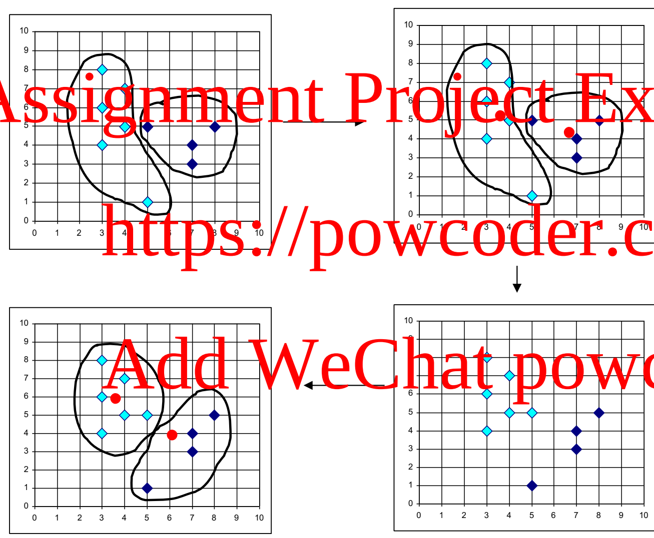
    for $i \in 1 \cdots N$ **do**

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

  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 train a classifier for each view
  - Each classifier predicts labels for a subset of the unlabeled instances, using only the features available in its view. These predictions are then used as ground truth to train the classifiers associated with the other views
  - 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: "Frustratingly simple" domain adaptation (Daumé III, 2007)

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

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

  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 annotated but can still produce the same level of accuracy
- Distant supervision: There is no labeled data, but you can generate some (potentially noisy) training data with some external resource such as a dictionary. For example, you can generate named entity annotation with a list of names.
- Multitask learning: The learning induces a representation that can be used to solve multiple tasks (learning POS tagging with syntactic parsing)