An Introduction To Natural Language Processing

Chapter 3: Nonlinear Text Classification

Jacob Eisenstein

Roadmap for this chapter

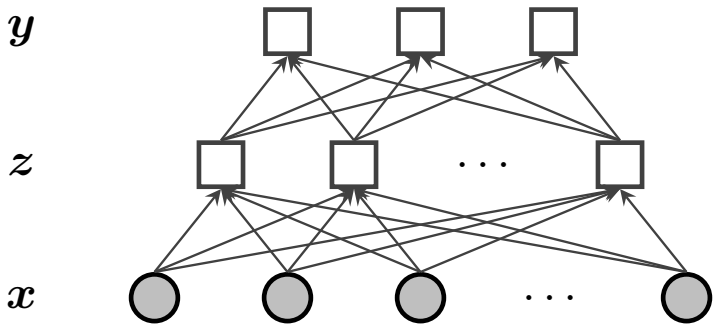
- Feedforward neural networks
  - Motivation
  - Activation functions
  - Inputs and outputs
- Learning neural networks
  - Backpropagation
  - Tricks of the trade
- Neural architectures for sequence data, part 1: convolution

A simple feedforward architecture

Suppose we want to label stories as $\mathcal{Y} = \{\textsc{Good}, \textsc{Bad}, \textsc{Okay}\}$.

- What makes a good story? Exciting plot; compelling characters; interesting setting ...
- Let's call this vector of features $\boldsymbol{z}$.
- If $\boldsymbol{z}$ is well-chosen, it will be easy to predict from $\boldsymbol{x}$ (the words), and it will make it easy to predict y (the label).

A simple feedforward architecture



Let's predict each z_k from $\boldsymbol{x}$ by binary logistic regression:

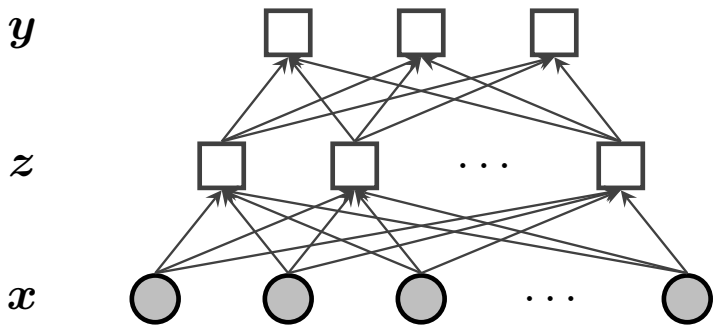
$$\begin{aligned}\Pr(z_k = 1 \mid \boldsymbol{x}) =& \sigma(\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}) \\ =& (1 + \exp(-\boldsymbol{\theta}_k^{(x \to z)} \cdot \boldsymbol{x}))^{-1}.\end{aligned}$$

The weights can be collected into a matrix,

$$\boldsymbol{\Theta}^{(x \to z)} = [\boldsymbol{\theta}_1^{(x \to z)}, \boldsymbol{\theta}_2^{(x \to z)}, \ldots, \boldsymbol{\theta}_{K_z}^{(x \to z)}]^\top,$$

so that $E[\boldsymbol{z}] = \sigma(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x})$, where σ is applied **elementwise**.

A simple feedforward architecture



Next we predict y from $\boldsymbol{z}$, again using logistic regression:

$$\begin{aligned}\Pr(y = j \mid z) &= \frac{\exp(\boldsymbol{\theta}_j^{(z \to y)} \cdot \boldsymbol{z} + b_j)}{\sum_{j' \in \mathcal{Y}} \exp(\boldsymbol{\theta}_{j'}^{(z \to y)} \cdot \boldsymbol{z} + b_{j'})},\end{aligned}$$

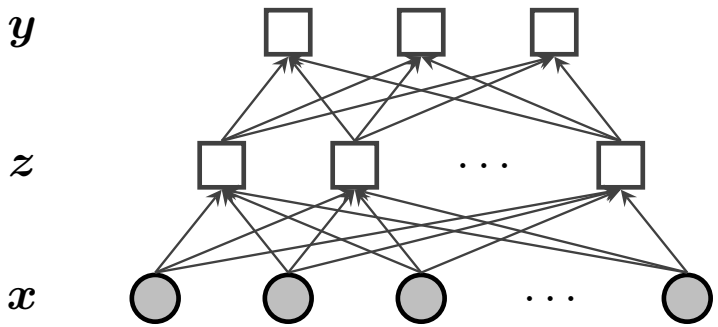
where each b_j is an offset. This is denoted,

$$p(\boldsymbol{y} \mid \boldsymbol{z}) = \text{SoftMax}(\boldsymbol{\Theta}^{(z \to y)} \boldsymbol{z} + \boldsymbol{b}).$$

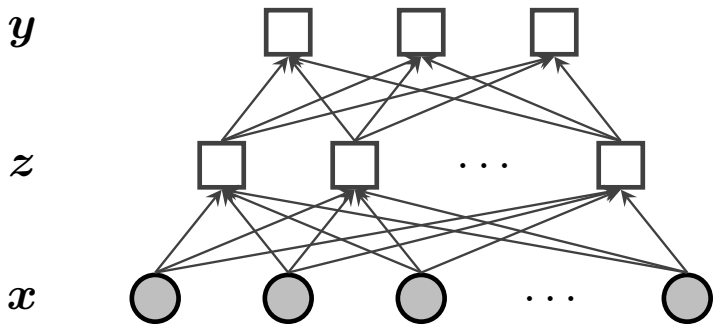
A simple feedforward architecture

To summarize:

$$\boldsymbol{z} = \sigma(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x})$$
$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{z}) = \mathrm{SoftMax}(\boldsymbol{\Theta}^{(z \to y)} \boldsymbol{z} + \boldsymbol{b}).$$



A simple feedforward architecture



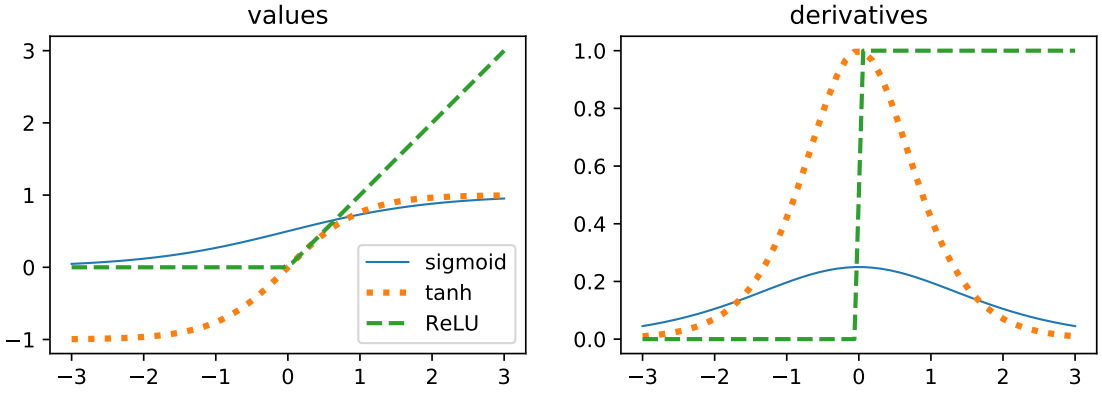
To summarize:

$$\boldsymbol{z} = \sigma(\boldsymbol{\Theta}^{(x \rightarrow z)} \boldsymbol{x})$$
$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{z}) = \mathrm{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z} + \boldsymbol{b}).$$

- In reality, we never observe $\boldsymbol{z}$, it is a **hidden layer**. We drop the expectation $E[\boldsymbol{z}]$, and compute $\boldsymbol{z}$ directly from $\boldsymbol{x}$.
- This makes $\mathrm{p}(y \mid \boldsymbol{x})$ a complex, nonlinear function of $\boldsymbol{x}$.
- We can have multiple hidden layers, $\boldsymbol{z}^{(1)}, \boldsymbol{z}^{(2)}, \ldots$, adding even more expressiveness.

Activation functions

The sigmoid in $\boldsymbol{z} = \sigma(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x})$ is an **activation function**.



In general, we write $\boldsymbol{z} = f(\boldsymbol{\Theta}^{(x \to z)} \boldsymbol{x})$ to indicate an arbitrary activation function. Other choices include:

- The **hyberbolic tangent** tanh, which is centered at zero, helping to avoid saturation.
- The **rectified linear unit** $\text{ReLU}(a) = \max(0, a)$, which is fast to evaluate and easy to analyze.

Outputs and loss functions

- The **softmax** output activation is used in combination with the negative log-likelihood loss, like logistic regression.
- In deep learning, this loss is called the **cross-entropy**:

$$\tilde{\boldsymbol{y}} = \text{SoftMax}(\boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z} + \boldsymbol{b})$$

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

where $\boldsymbol{e}_{y^{(i)}} = [0, 0, \ldots, 0, \underbrace{1}_{y^{(i)}}, 0, \ldots, 0]$, a **one-hot vector**.

- A margin loss can also be used, with $\Psi(y; \boldsymbol{x}^{(i)}) = \boldsymbol{\Theta}^{(z \rightarrow y)} \boldsymbol{z} + b$.

Gradient descent in neural networks

In general, neural networks are learned by gradient descent, typically with minibatches.

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

where:

- $\eta^{(t)}$ is the learning rate at update t;
- $\ell^{(i)}$ is the loss on instance (minibatch) i; and
- $\nabla_{\boldsymbol{\theta}_k^{(z \to y)}} \ell^{(i)}$ is the gradient of the loss with respect to the column vector of output weights $\boldsymbol{\theta}_k^{(z \to y)}$,

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

Backpropagation

If we don't observe $\boldsymbol{z}$, how can we learn the weights $\boldsymbol{\Theta}^{(x \rightarrow z)}$?
Backpropagation: compute a loss on $\boldsymbol{y}$, and apply the chain rule of calculus to compute a gradient on all parameters:

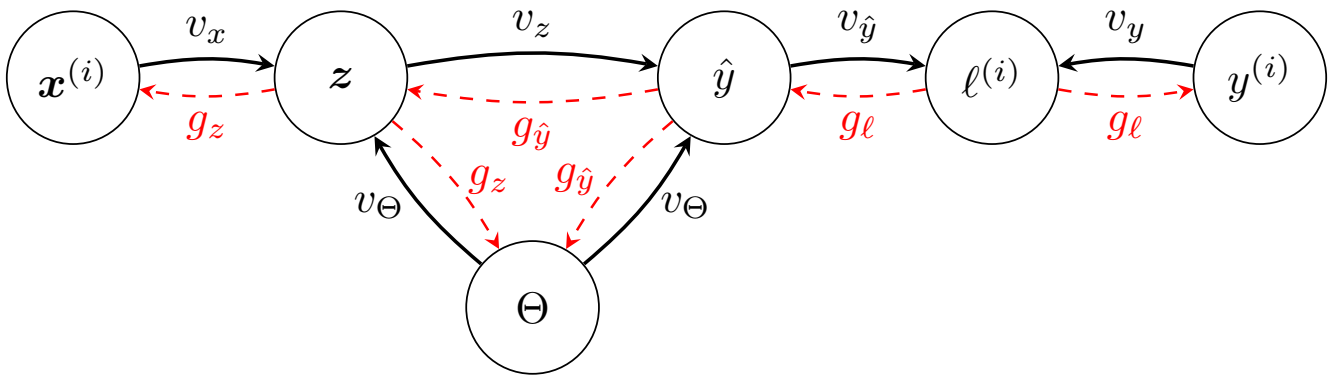
$$\begin{aligned}
\frac{\partial \ell^{(i)}}{\partial \theta_{n,k}^{(x \rightarrow z)}} &= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial z_k}{\partial \theta_{n,k}^{(x \rightarrow z)}} \\
&= \frac{\partial \ell^{(i)}}{\partial z_k} \frac{\partial f(\boldsymbol{\theta}_k^{(x \rightarrow z)}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

When f is sigmoid, $f'(\boldsymbol{\theta}_k^{(x \rightarrow z)} \cdot \boldsymbol{x}) = z_k \times (1 - z_k)$.

Backpropagation as an algorithm

Construct a directed **computation graph**, with nodes for inputs, outputs, hidden layers, and parameters. This graph must be acylic.

- **Forward pass**: values (e.g., v_x) go from parents to children.
- **Backward pass**: gradients (e.g., g_z) go from children to parents, implementing the chain rule of calculus.



Given the gradients, perform gradient descent.

"Tricks" for better performance

- Preventing overfitting with regularization and dropout
- Smart initialization
- Online learning

Tricks: regularization and dropout

Because neural networks are powerful learners, overfitting is a potential problem.

- **Regularization** works similarly for neural nets as it does in linear classifiers: penalize the weights by $\lambda \sum_{i,j} \theta_{i,j}^2$. This is sometimes called "weight decay."
- **Dropout** prevents overfitting by randomly deleting weights or nodes during training. This prevents the model from relying too much on individual features or connections.
- Dropout rates are usually between 0.1 and 0.5, tuned on validation data.

Tricks: initialization

Unlike linear classifiers, neural networks usually optimize a **non-convex** objective. This means that initialization can affect the outcome.

- If the initial weights are too large, activations may saturate the activation function (for sigmoid or tanh) or overflow (for ReLU activation).
- If they are too small, learning may take too many iterations to converge.

This motivates random initialization, but at what scale?

Tricks: random initial weights

Initializations can be derived from the premise that the variance of the initial gradients should be constant throughout the network:

- For tanh activations, **Xavier initialization** samples sampling initial weights from a uniform distribution whose range depends on the in-degree and out-degree.[1]
- For ReLU activations, He et al. (2015) propose sampling from a zero mean Gaussian, with variance that is a decreasing function of the in-degree.

Alternatively, weight matrices can be initalized to be **orthonormal** ($\Theta^\top \Theta = \mathbb{I}$), so that the norm of the activations is preserved, $||\Theta \boldsymbol{x}|| = ||\boldsymbol{x}||$.[2]

[1] Glorot, Bordes, and Bengio 2011.

[2] Saxe, McClelland, and Ganguli 2014.

Tricks: online learning

Stochastic gradient descent is the simplest learning algorithm for neural networks, but there are many other choices:

- AdaGrad and Adam use **adaptive learning rates** for each parameter.
- Nesterov Accelerated Gradient uses **momentum** to "roll" through flat regions of the objective function.
- In practice, most implementations **clip** gradients to some maximum magnitude before making updates.

Early stopping: check performance on a development set, and stop training when performance starts to get worse.

Neural architectures for sequence data

Text is naturally viewed as a sequence of tokens, $w_1, w_2, \ldots, w_M$.

- Context is lost when this sequence is converted to a bag-of-words.
- Instead, a **lookup layer** can compute **embeddings** for each token, resulting in a matrix $\mathbf{X}^{(0)} = \mathbf{\Theta}^{(x \to z)}[\boldsymbol{e}_{w_1}, \boldsymbol{e}_{w_2}, \ldots, \boldsymbol{e}_{w_M}]$, where $\mathbf{X}^{(0)} \in \mathbb{R}^{K_e \times M}$.
- Higher-order representations $\mathbf{X}^{(d>0)}$ can then be computed from $\mathbf{X}^{(0)}$, as shown on the next slide.
- For classification, the top layer $\mathbf{X}^{(D)}$ must be converted into a vector, using a **pooling** operation, such as max-pooling:

$$z_k = \max(x_{k,1}^{(D)}, x_{k,2}^{(D)}, \ldots, x_{k,M}^{(D)}). \tag{1}$$

Convolutional neural networks

Convolutional neural networks compute successively higher representations $\mathbf{X}^{(d)}$ by convolving $\mathbf{X}^{(d-1)}$ with a set of local filter matrices $\mathbf{C}$:

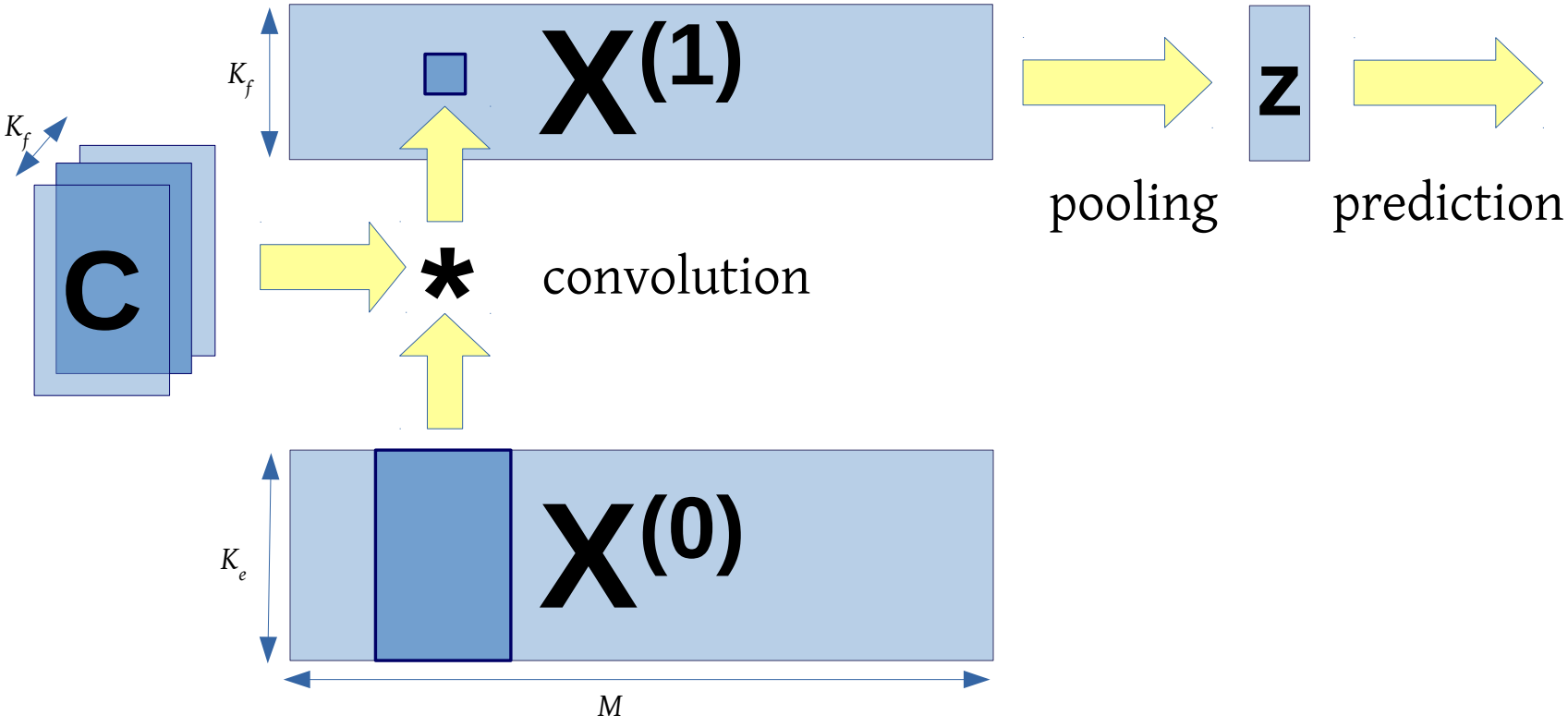
$$x_{k,m}^{(d)} = f\left(b_k + \sum_{k'=1}^{K_e} \sum_{n=1}^{h} c_{k',n}^{(k)} \times x_{k',m+n-1}^{(d-1)}\right),$$

where

- f is a non-linear activation function;
- h is the filter size;
- the filter parameters c are learned from data.

In this way, each $x_{k,m}^{(d)}$ is a function of locally adjacent features at the previous level.

Convolutional neural networks



Neural architectures for sequence data

- Convolutional neural networks are sensitive to local dependencies between words.
- In **recurrent neural networks** (chapter 5), a model of context is constructed while processing the text from left-to-right. These networks are theoretically sensitive to global dependencies.
- In **self-attentional networks** (chapter 18), the model of context is adaptive: each word can choose its own context.
- In 2019, this remains an active area of research!

Roadmap for this chapter

- Feedforward neural networks
  - Motivation
  - Activation functions
  - Inputs and outputs
- Learning neural networks
  - Backpropagation
  - Tricks of the trade
- Neural architectures for sequence data, part 1: convolution

References I

- Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep Sparse Rectifier Networks". In: *Proceedings of Artificial Intelligence and Statistics (AISTATS)*, pp. 315–323.
- He, Kaiming et al. (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the International Conference on Computer Vision (ICCV)*, pp. 1026–1034.
- Saxe, Andrew M, James L McClelland, and Surya Ganguli (2014). "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks". In: *Proceedings of the International Conference on Learning Representations (ICLR)*.