Chapter 10: Deep Learning

Yonghyun Kwon

Department of Mathematics, Korea Military Academy

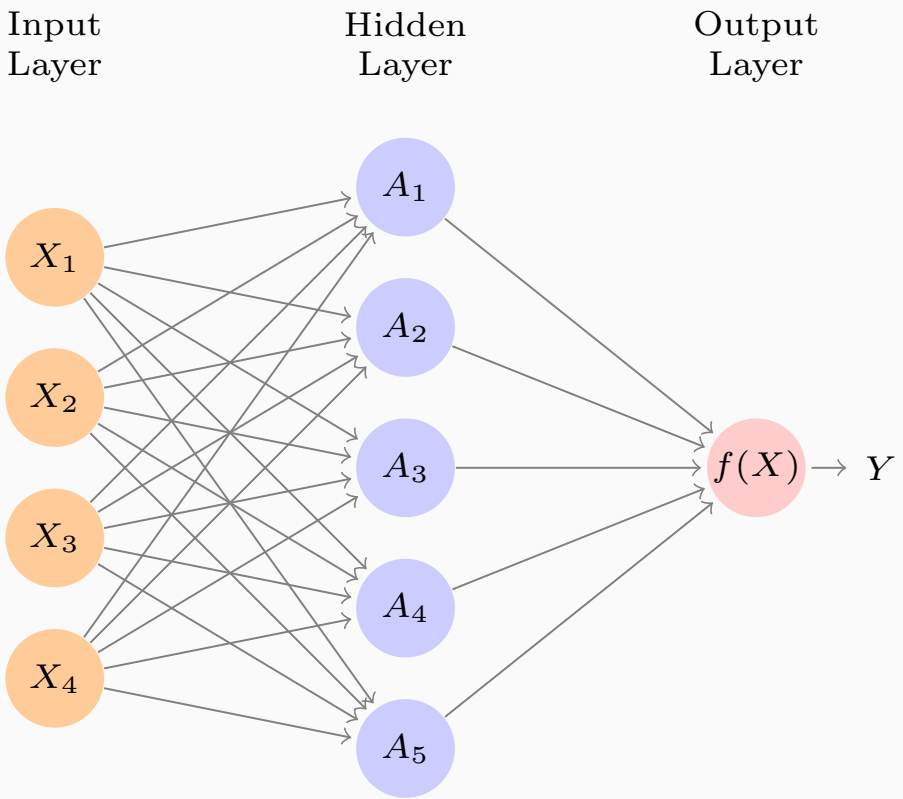
Single Layer Neural Networks

Deep Learning

- Originated from **neural networks** popular in the 1980s.
- Interest declined in the 1990s with methods like **SVMs** and **Boosting**.
- Re-emerged around **2010** as *Deep Learning*—multi-layer neural networks trained on large datasets.
- Success driven by advances in **computing power**, **big data**, and **software** (e.g., TensorFlow, PyTorch).

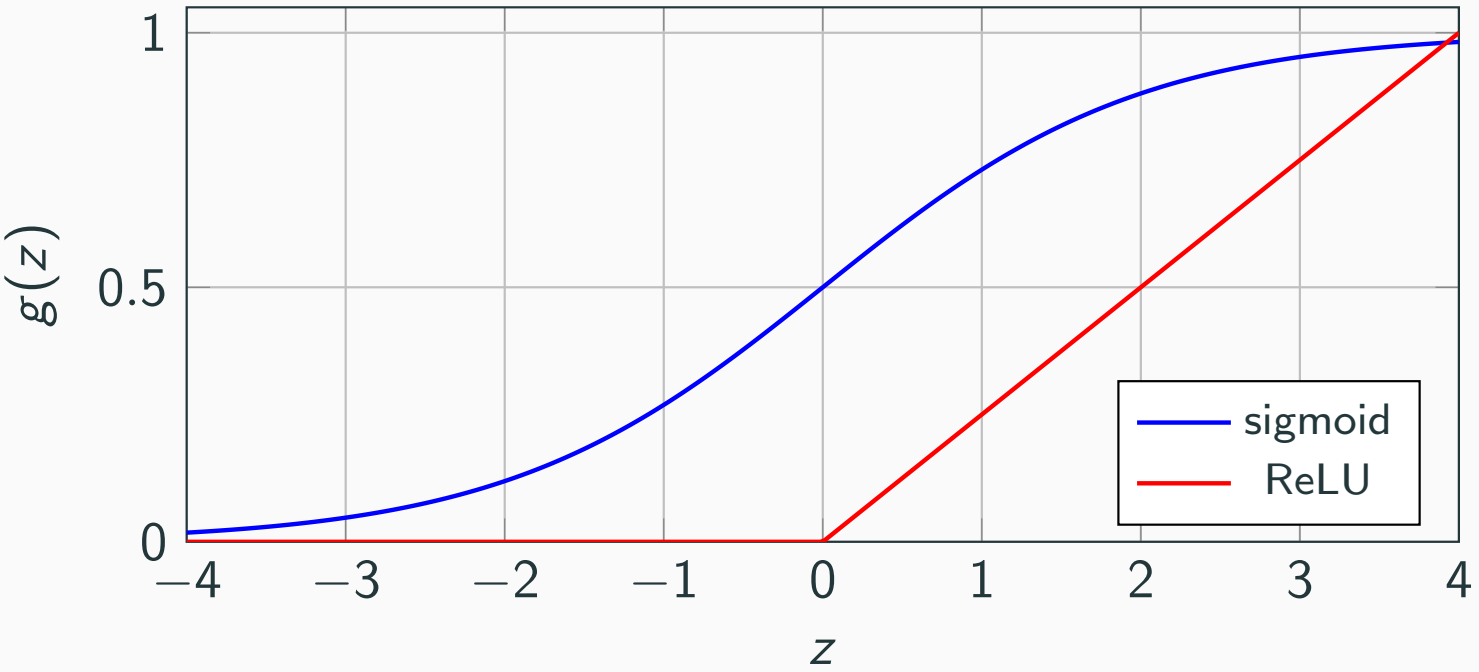
(Feede-Forward) Single-Layer Neural Network

$$f(\mathbf{X}) = \beta_0 + \sum_{k=1}^{K} \beta_k \, h_k(\mathbf{X}) = \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right).$$



Activations and Activation Functions

- $A_k = h_k(\mathbf{X}) = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$ are the *activations* in the hidden layer.
- $g(z)$ is the *activation function*.
  - Popular choices: *sigmoid* and *rectified linear unit(ReLU)*.



Activation Functions: Sigmoid and ReLU

Sigmoid activation:

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

It is smooth and differentiable, and maps any real number to $(0, 1)$.

ReLU (Rectified Linear Unit):

$$g(z) = (z)_+ = \begin{cases} 0, & z < 0, \\ z, & z \geq 0. \end{cases}$$

ReLU introduces nonlinearity while keeping piecewise linear form.

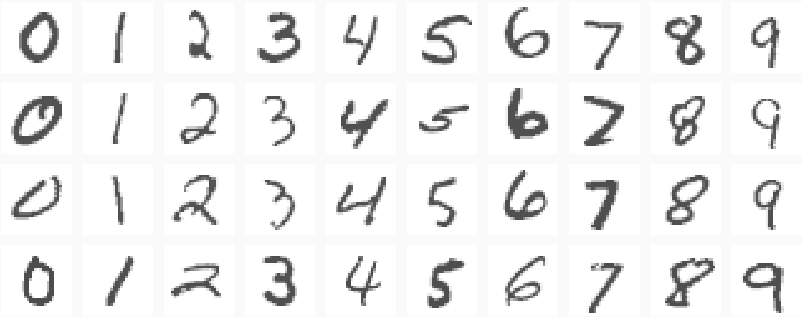
Activations and Activation Functions

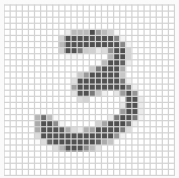
- Hidden-layer activations are typically **nonlinear**; otherwise the neural network collapses to a linear model.
- For regression, **squared error loss** is typically used to fit a neural network:

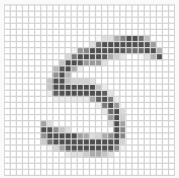
$$\sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i)\right)^2.$$

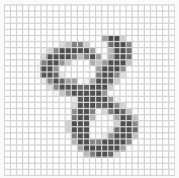
Multilayer Neural Networks

Example: MNIST handwritten digits



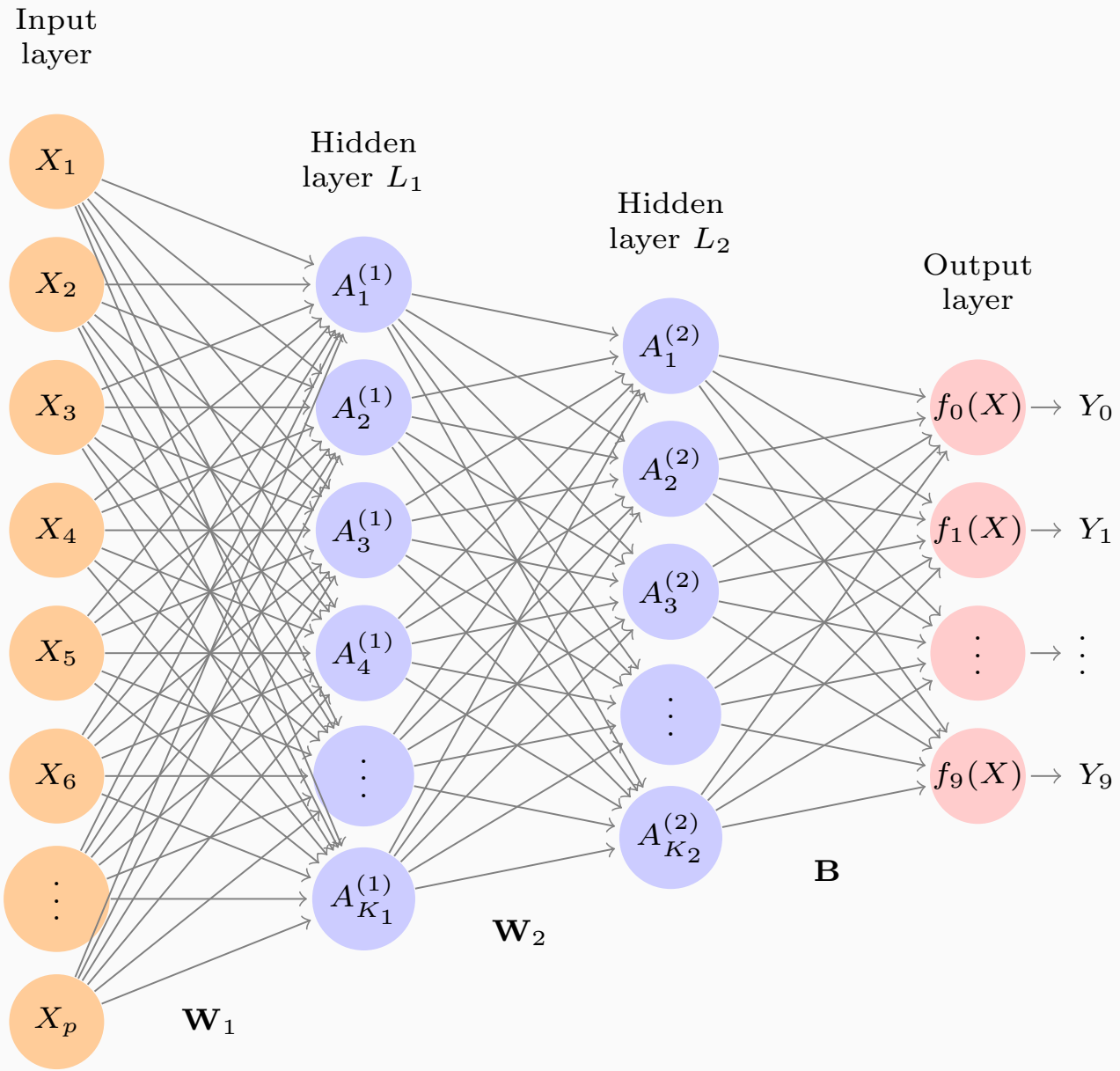






- Handwritten digits: 28×28 grayscale images.
- Features: $p = 28 \times 28 = 784$ pixel values; labels: classes 0–9.
- **Goal:** build a classifier to predict the image class.
- Two layers: $K_1 = 256$ units (layer 1), $K_2 = 128$ units (layer 2).
- Including *biases*, the number of parameters(*weights*) is:

$$784 \times 256 + 256 \;+\; 256 \times 128 + 128 \;+\; 128 \times 10 + 10 \;=\; \mathbf{235{,}146}.$$


Input layer
Hidden layer L_1
Hidden layer L_2
Output layer
X_1
X_2
X_3
X_4
X_5
X_6
X_p
$A_1^{(1)}$
$A_2^{(1)}$
$A_3^{(1)}$
$A_4^{(1)}$
$A_{K_1}^{(1)}$
$A_1^{(2)}$
$A_2^{(2)}$
$A_3^{(2)}$
$A_{K_2}^{(2)}$
$f_0(X) \rightarrow Y_0$
$f_1(X) \rightarrow Y_1$
$f_9(X) \rightarrow Y_9$
$\mathbf{W}_1$
$\mathbf{W}_2$
$\mathbf{B}$

Details of the Output Layer

Let $Z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}$, $m = 0, \ldots, 9$ be 10 linear combinations of activations at second layer. To represent class probabilities, we use *softmax* function:

$$f_m(\mathbf{X}) = \Pr(Y = m \mid \mathbf{X}) = \frac{e^{Z_m}}{\sum_{r=0}^{9} e^{Z_r}}$$

Fit by minimizing *cross-entropy (negative multinomial log-likelihood):*

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log f_m(\mathbf{x}_i)$$

$y_{im} = 1$ if the true class for observrvation i is m, else 0: *one-hot encoded*.

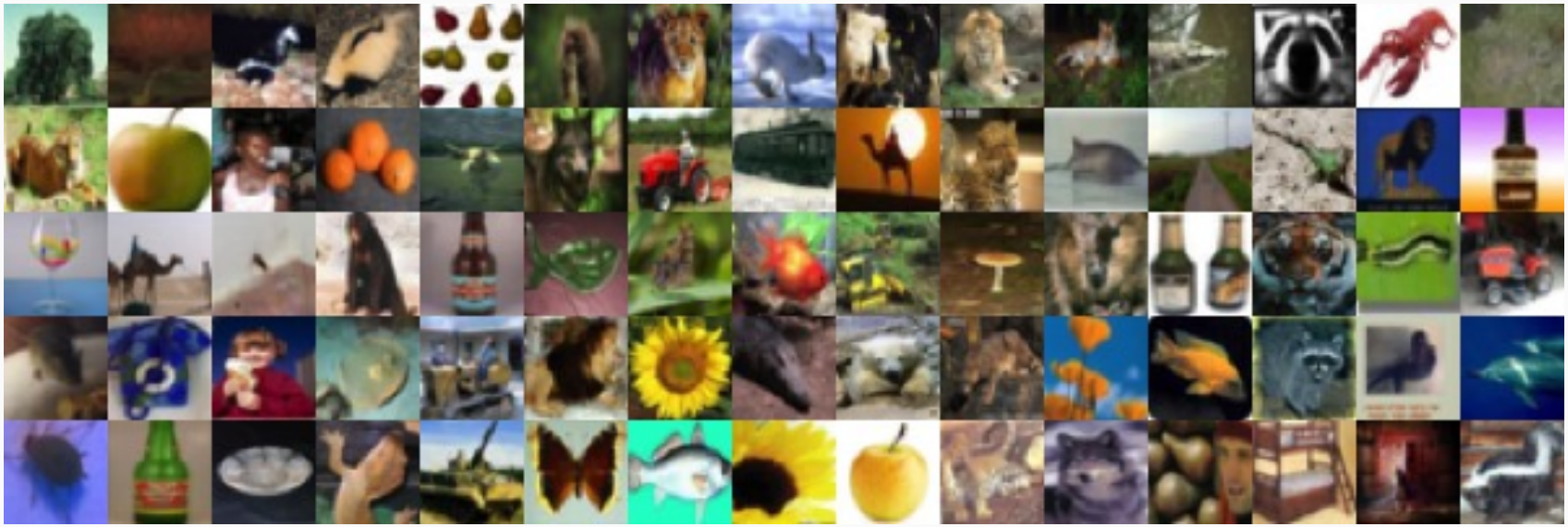
MNIST: Test Error

Method	Test Error
Neural Net + Ridge	2.3%
Neural Net + Dropout	1.8%
Multinomial Logistic Regression	7.2%
Linear Discriminant Analysis	12.7%

- With many parameters, **regularization** is essential.
- Best reported rates are $< 0.5\%$.
- Human error is about 0.2% on 10K test images.

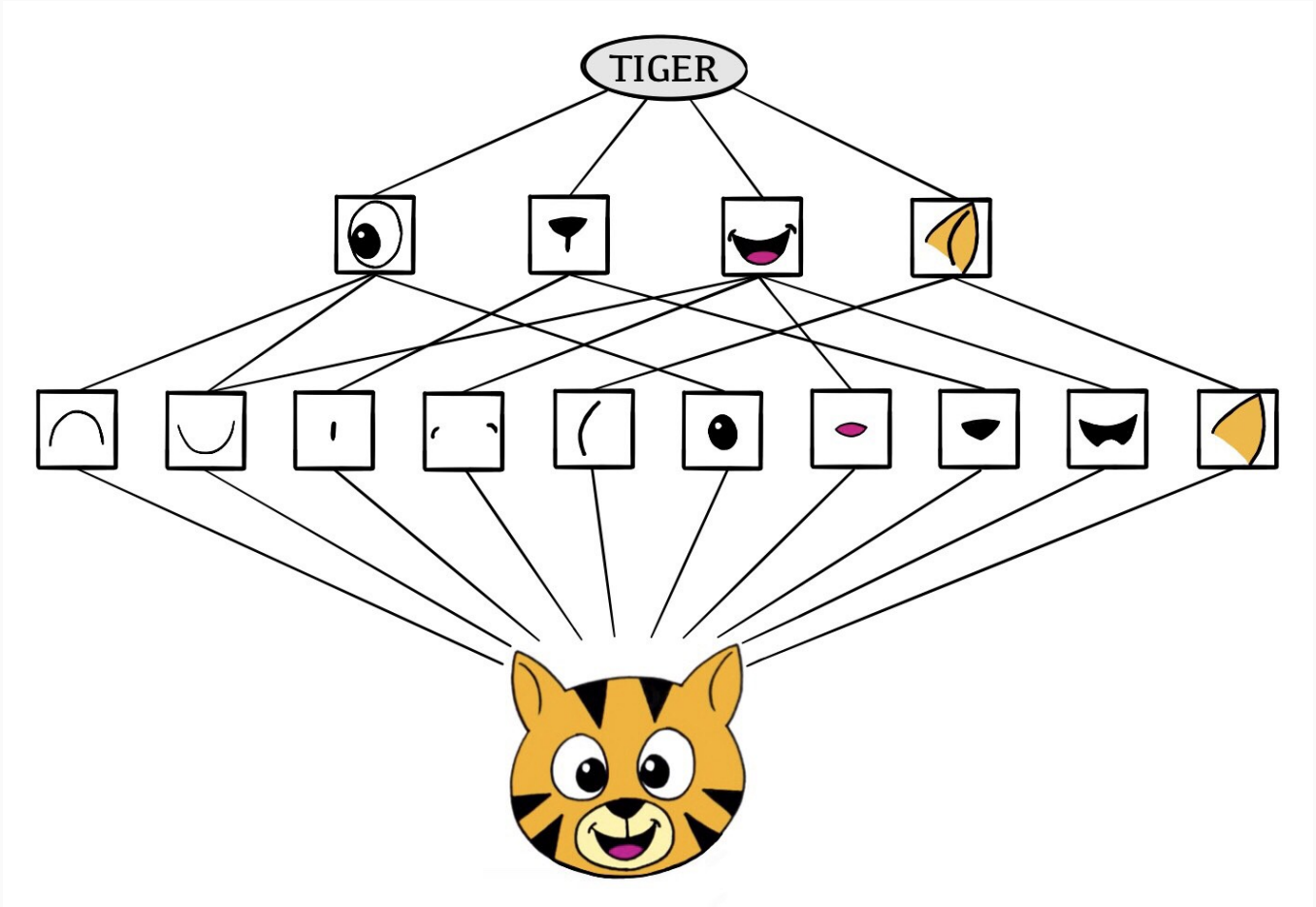
Convolutional Neural Networks

Convolutional Neural Networks (CNN)



- Major success story for image classification.
- Images are 3D *feature maps*: width $\times$ height $\times$ *channels* (RGB).
- Example: `CIFAR-100`: colored natural images with 100 classes.
  - $32 \times 32 \times 3$ array of eight-bit numbers.

How CNNs Work



CNNs learn **hierarchical** features: (e.g. edges $\rightarrow$ shapes $\rightarrow$ objects) using *convolution* + *pooling* layers.

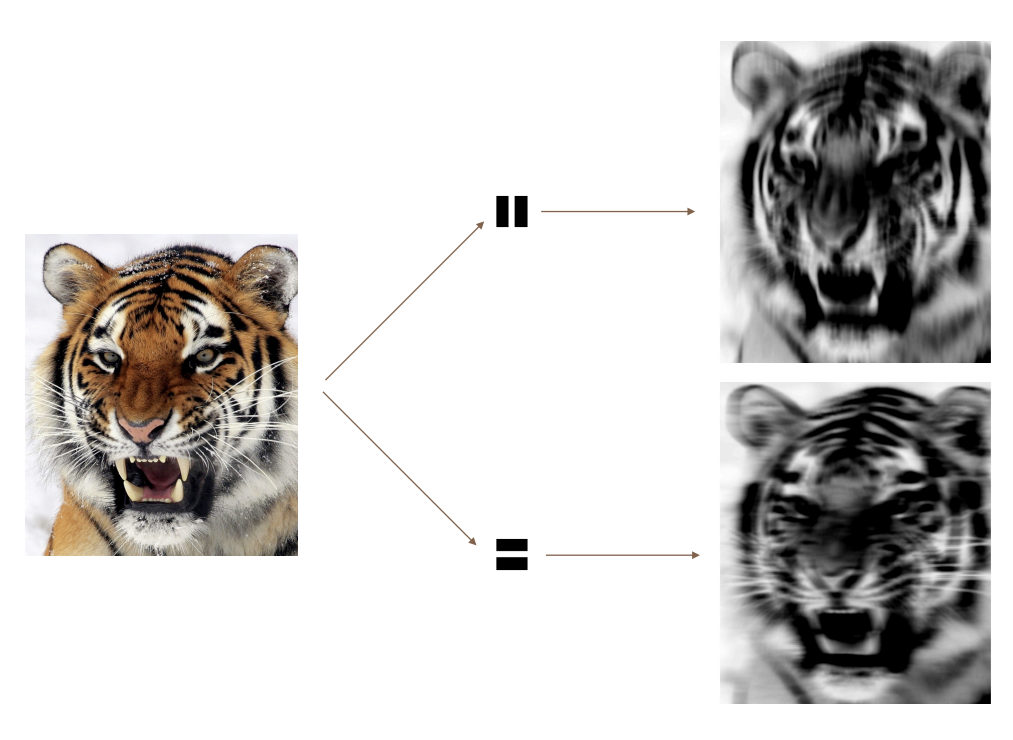
Convolution

$$\text{Input image: } \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \quad \text{Convolution Filter: } \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

$$\Rightarrow \text{Convolved Image: } \begin{bmatrix} a\alpha+b\beta+d\gamma+e\delta & b\alpha+c\beta+e\gamma+f\delta \\ d\alpha+e\beta+g\gamma+h\delta & e\alpha+f\beta+h\gamma+i\delta \\ g\alpha+h\beta+j\gamma+k\delta & h\alpha+i\beta+k\gamma+l\delta \end{bmatrix}$$

- The *convolution* is done via **dot-products**
- The convolved image highlights regions of the original image that resemble the convolution filter.

Convolution example



- Two filters—vertical and horizontal—are applied to the tiger image.
- They highlight vertical and horizontal stripes respectively.
- The image serves as the input layer, and the filtered outputs as the first hidden layer.

Pooling

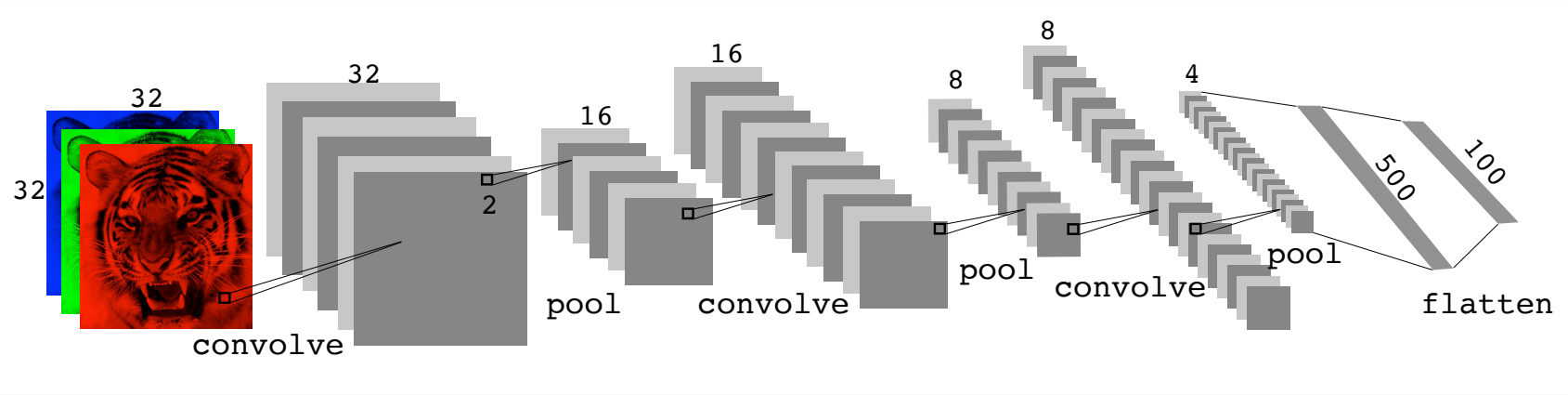
$$\text{Max pool} \quad \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

- *Pooling* condenses a large image into a smaller summary image.
- *Max pooling* summarizes each non-overlapping 2×2 block by its maximum value.
- This reduces image size by a factor of two in each direction.
- Provides some **location invariance**.

Source: attached slides.

:contentReference[oaicite:5]index=5

CNN Architecture



- A *convolution layer* uses multiple filters, each producing a 2D feature map (The number of filters = number of channels).
- A *pooling layer* reduces spatial dimensions for each channel.
- Deep CNNs repeat this convolve–then–pool sequence many times.
- As spatial size decreases, the number of filters usually increases.
- After several layers, feature maps are *flattened* and passed to fully connected layers, ending with a **softmax output**.

Data Augmentation

- *Data augmentation* replicates training images with random natural distortions.
- Common transformations: zoom, shift, rotation, shear, and horizontal flip.
- Expands the training set and helps **prevent overfitting**.

Document Classification: IMDB Movie Reviews

- The `IMDB corpus` consists of user movie reviews labeled for **sentiment** as **positive** or **negative**.
- Example:

  This has to be one of the worst films of the 1990s... everyone else in the theater just started talking to each other, leaving or crying into their popcorn.

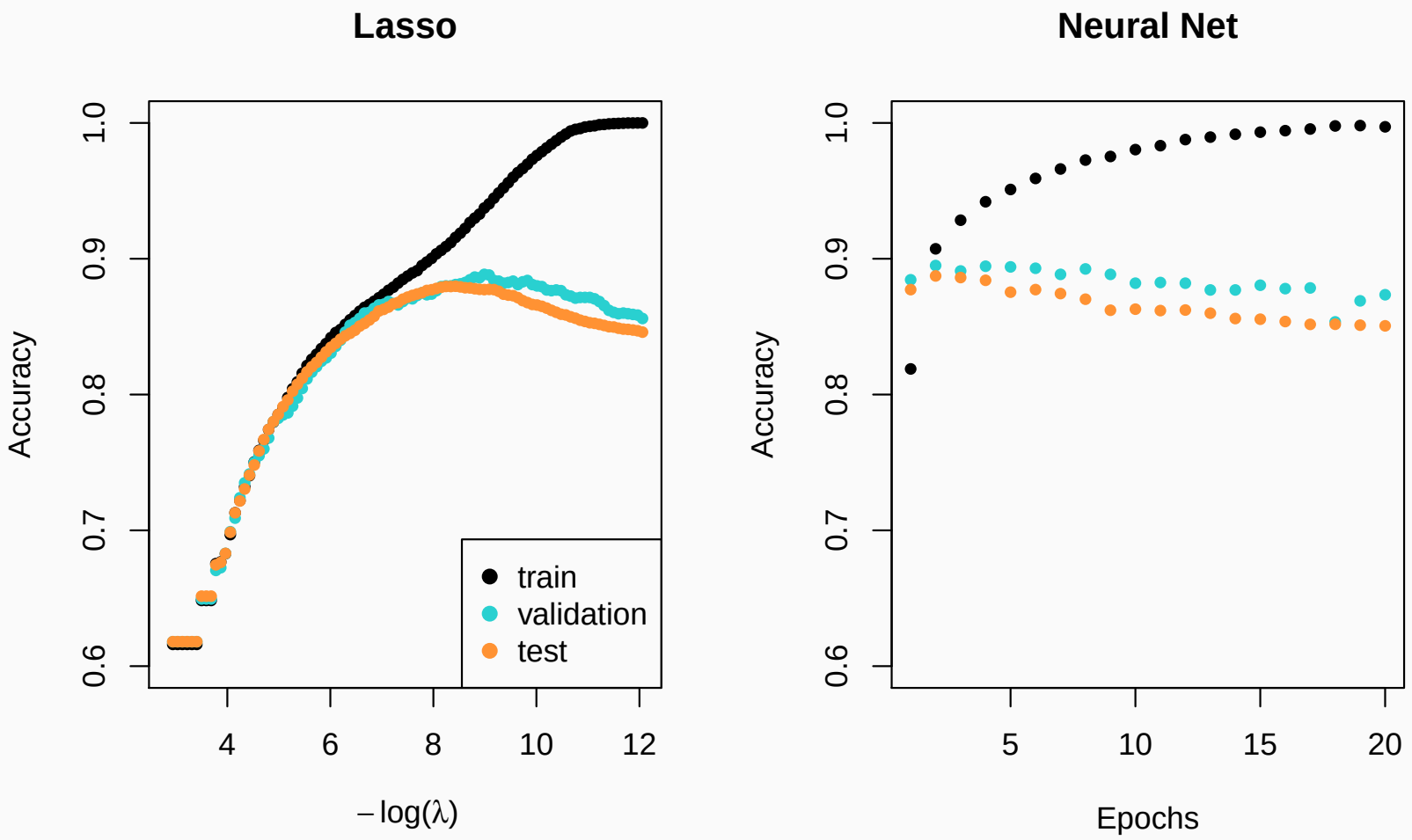
- **Goal**: build a classifier to predict the sentiment of a review.

Featurization: Bag-of-Words

Data Before Bag-of-Words	**Data After Applying Bag-of-Words**				
Text Documents	**loved**	**movie**	**great**	**awful**	**...**
I loved this movie! It was great, great, great.	1	1	1	0	...
This movie was so awful!	0	1	0	1	...
What was this movie even about?	0	1	0	0	...

- Documents vary in length and word order; we need to construct features $\mathbf{X}$.
- From a dictionary, select the 10K most frequent words.
- For each document, create a binary vector of length $p = 10{,}000$: 1 if the word occurs, 0 otherwise.
- Bag-of-words uses *unigrams*; we can also use *bigrams* (word pairs).

Lasso vs Neural Network — IMDB Reviews



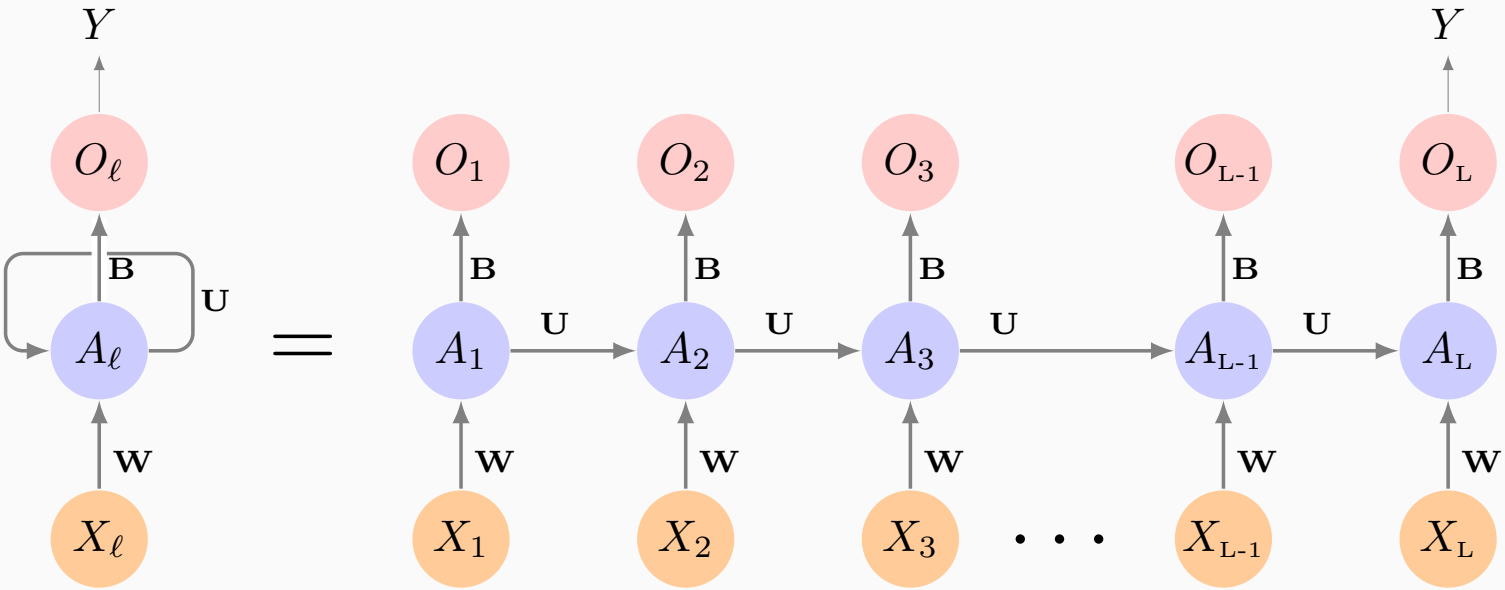
Lasso logistic regression performs as well as the neural network.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs)

- Designed for data that occur as **sequences**:
  - text, time series, speech, or handwriting.
- Capture **temporal dependencies** by remembering past information.
- Input features form a sequence of vectors $\mathbf{X} = \{X_1, X_2, \ldots, X_L\}$.
- Output Y can be a single target (e.g., sentiment) or a sequence (e.g., translated text).

Simple RNN Architecture



- Hidden layer is a sequence of vectors A_ℓ that receives input X_ℓ and previous state $A_{\ell-1}$.
- **Same** weights $(\mathbf{W}, \mathbf{U}, \mathbf{B})$ are reused at each time step — *recurrent*.
- The hidden sequence $\{A_\ell\}$ evolves as each element X_ℓ is processed.

RNN Computation

$$A_{\ell k} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\right), \quad O_\ell = \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k}.$$

- The hidden state combines current input and past activation.
- We often use only the final output O_L for prediction.
- Model is trained by minimizing squared error (or cross-entropy) over all sequences.

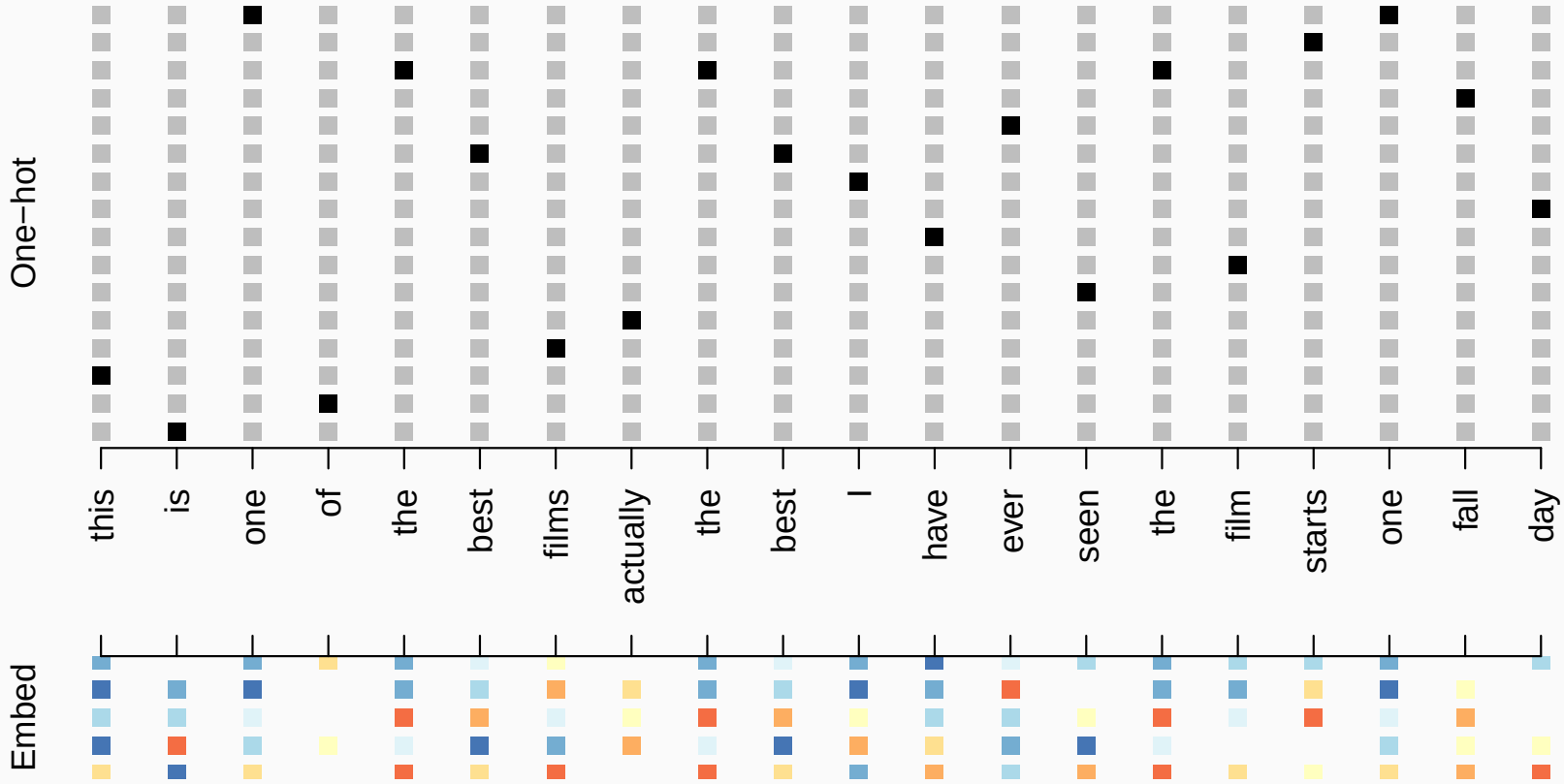
$$\sum_{i=1}^{n} (y_i - o_{iL})^2 = \sum_{i=1}^{n} \left(y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k\, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s}\right)\right)\right)^2$$

RNN for IMDB Reviews

Dictionary			One-hot
1	Money		$\{1, 0, 0, \ldots, 0\}$
2	Coin		$\{0, 1, 0, \ldots, 0\}$
3	Sale	$\Longrightarrow$	$\{0, 0, 1, \ldots, 0\}$
$\vdots$	$\vdots$		$\vdots$
100	Economy		$\{0, 0, 0, \ldots, 1\}$

- Each document is a sequence of words $\{W_\ell\}_1^L$ (truncate/pad to $L = 500$ words).
- Each word W_ℓ is one-hot encoded as a binary vector of length 10K.
- One-hot vectors are sparse and high-dimensional.
- Use a pretrained *word embedding matrix* E ($m \times 10\text{K}$) to map each word to an m-dimensional vector ($m \ll 10\text{K}$).

Word Embedding

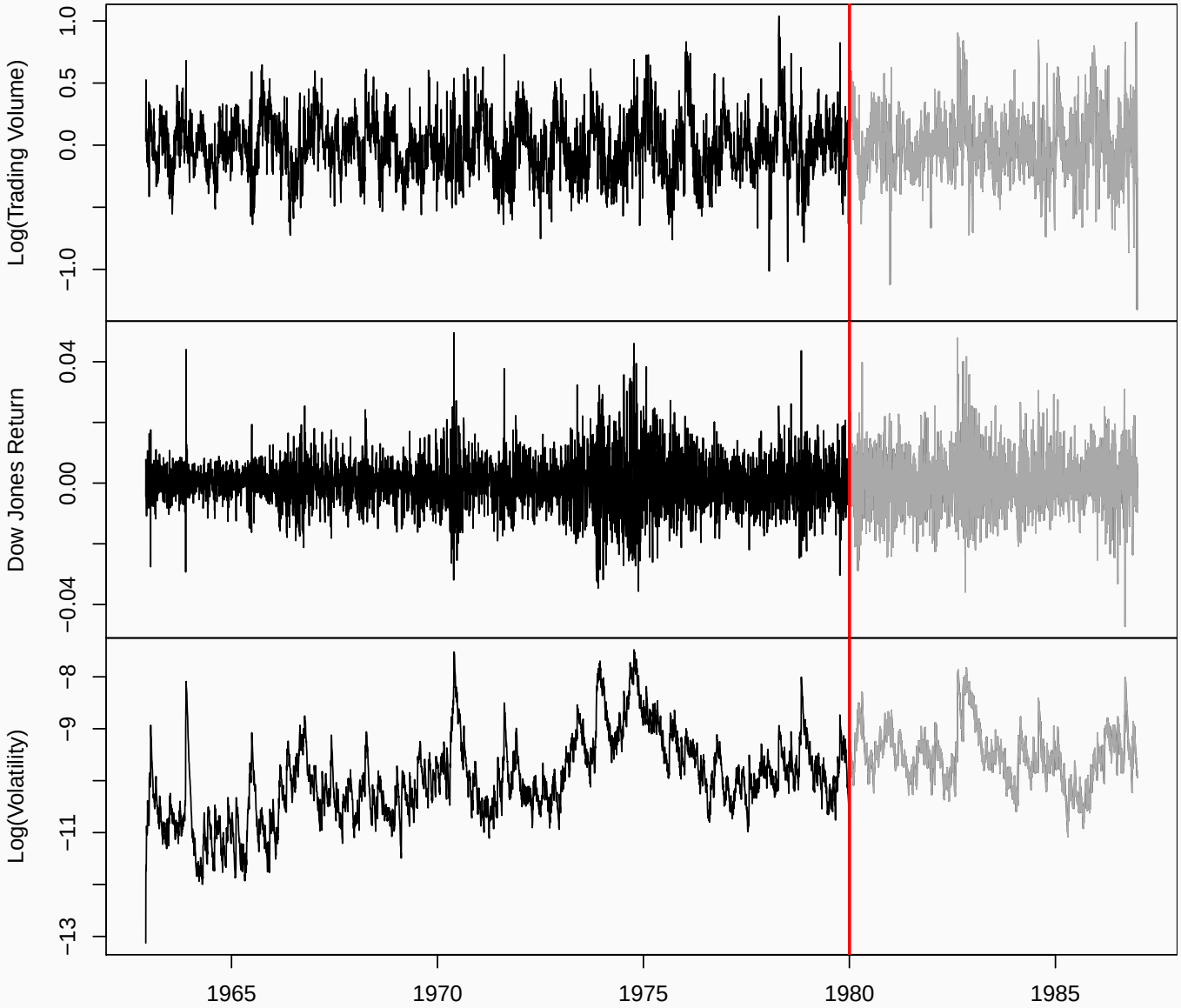


Common methods: **word2vec** and **GloVe**.

RNN on IMDB Reviews (Results)

- A simple RNN achieved only about **76% accuracy**.
- Using a more advanced *LSTM (Long Short-Term Memory)* network improved performance to **87%**.
- LSTM adds both short- and long-term memory connections.
  - A_ℓ receives input from $A_{\ell-1}$ (*short-term memory*) as well as from a version that reaches further back in time(*long-term memory*).

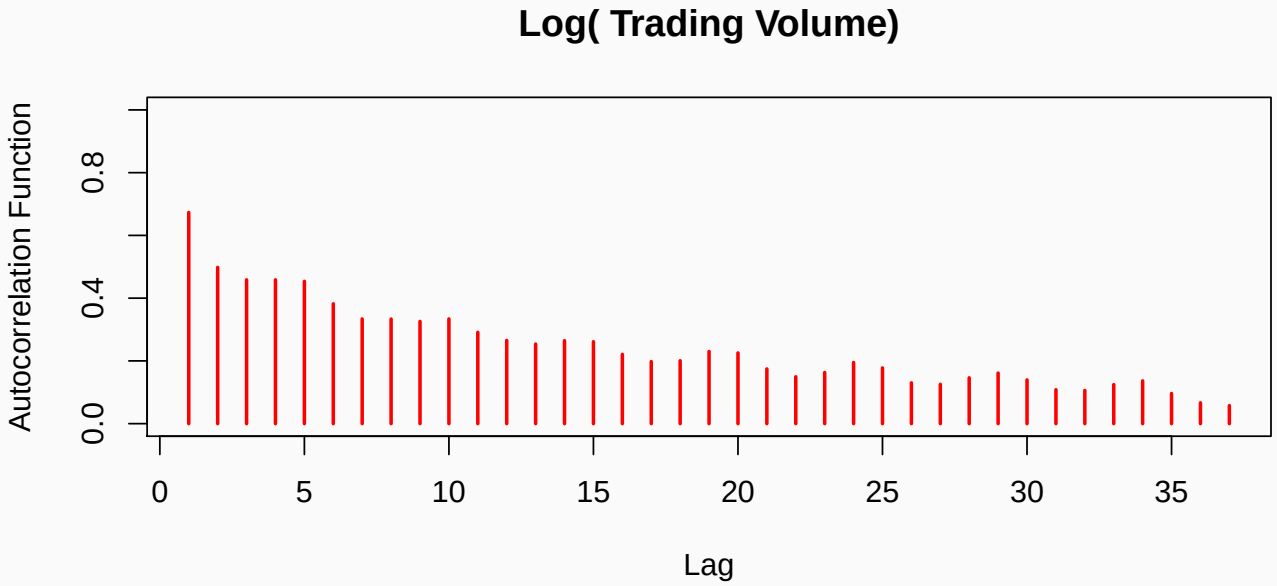
Time Series Forecasting



NYSE Data Description

- Period: Dec 3 1962 – Dec 31 1986 (6,051 trading days).
- Features:
  - **Log trading volume**(v_t) — fraction of shares traded (relative to 100-day moving average).
  - **Dow Jones return**(r_t) — change in log index between consecutive days.
  - **Log volatility**(z_t) — absolute daily price movement (log scale).
- Goal: predict tomorrow's **log trading volume**(v_{t+1}) from past values of these variables.

Autocorrelation



- The *autocorrelation* at lag ℓ is the correlation of $(v_t, v_{t-\ell})$.
- Strong positive correlations indicate that past values help predict future values.
- The response variable v_t is also a predictor in the future.

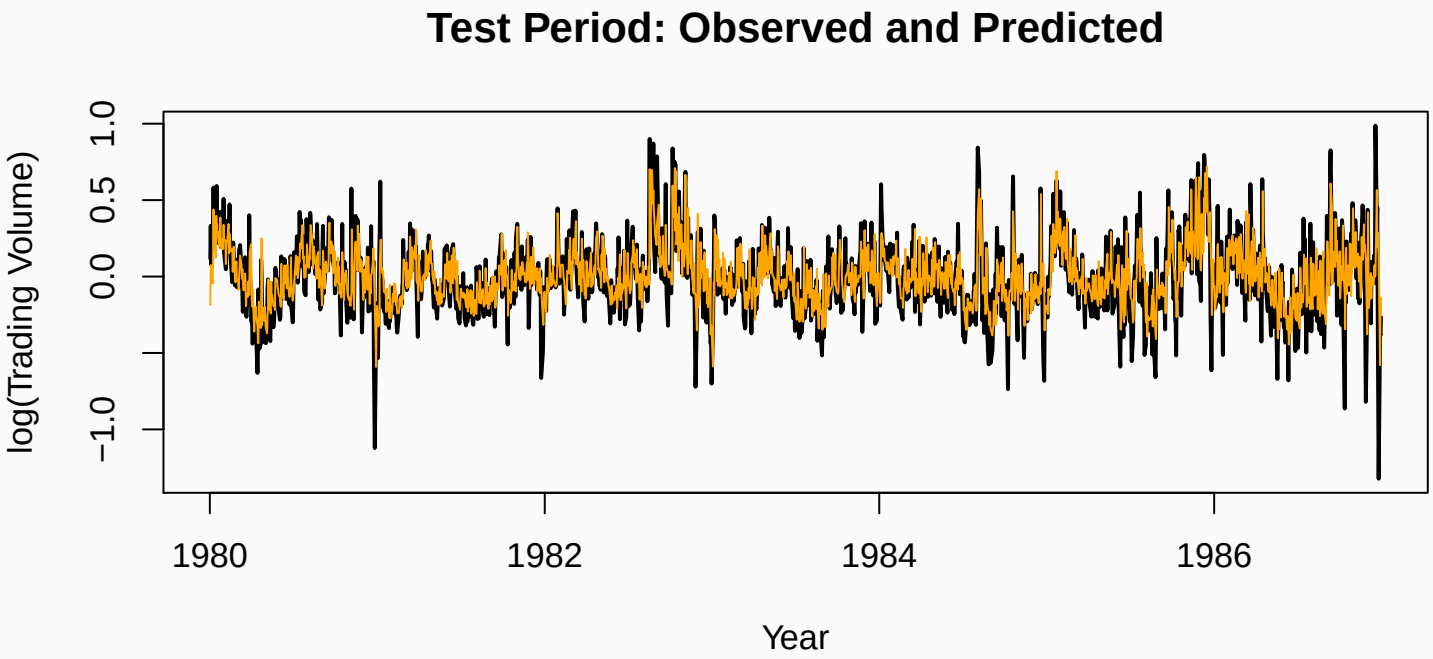
RNN Forecaster

- We only have one series of data. How do we set up RNN?
- We extract many short subsequences of length L (the *lag*).
- For each target day t, define

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \; X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \; \cdots, \; X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \; Y = v_t.$$

- With $T = 6{,}051$ and $L = 5$, we obtain 6,046 training examples (X, Y).
- First 4,281 used for training; next 1,770 for testing.
- RNN fitted with 12 hidden units per lag step A_ℓ.

RNN Results for NYSE Data



Black - true volumes; orange - predicted.

- $R^2 = 0.42$ for the RNN model.
- Baseline ("straw man") using yesterday's value gives $R^2 = 0.18$.

Autoregression (AR) Forecaster

- The RNN structure resembles an *autoregression(AR)* model.
- For an order-L AR model:

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \hat{\beta}_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}.$$

- Can include lagged values of Dow Jones return and log volatility as predictors.

Autoregression Results for NYSE Data

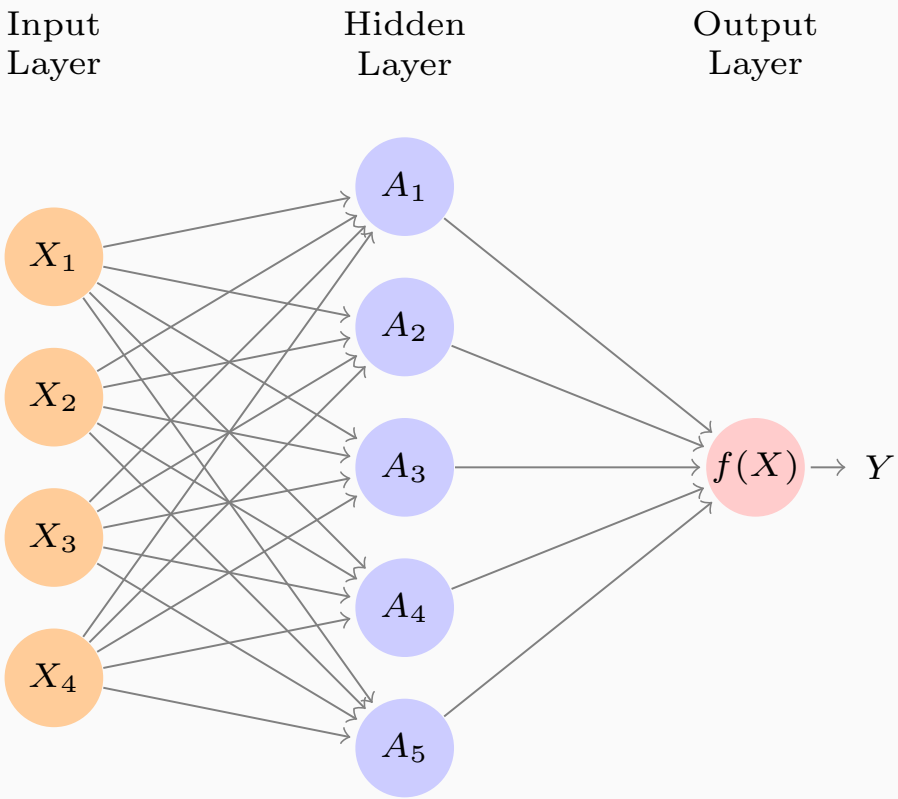
- $R^2 = 0.41$ for AR(5) model (16 parameters).
- $R^2 = 0.42$ for RNN model (205 parameters).
- $R^2 = 0.42$ for AR(5) fitted by neural network.
- Including `day of week`(Monday, Tuesday, etc) improves all models to $R^2 = 0.46$.

Summary of Recurrent Neural Networks

- Many extensions of RNNs improve sequence modeling.
- **1D CNNs** can model sequences by sliding filters along embeddings.
- **Bidirectional** and multi-layer RNNs capture richer context.
- *Seq2Seq* models handle input–output sequences (e.g., translation).
- Modern systems (e.g., **Google Translate**) use large-scale, optimized RNN architectures.

Fitting a Neural Network

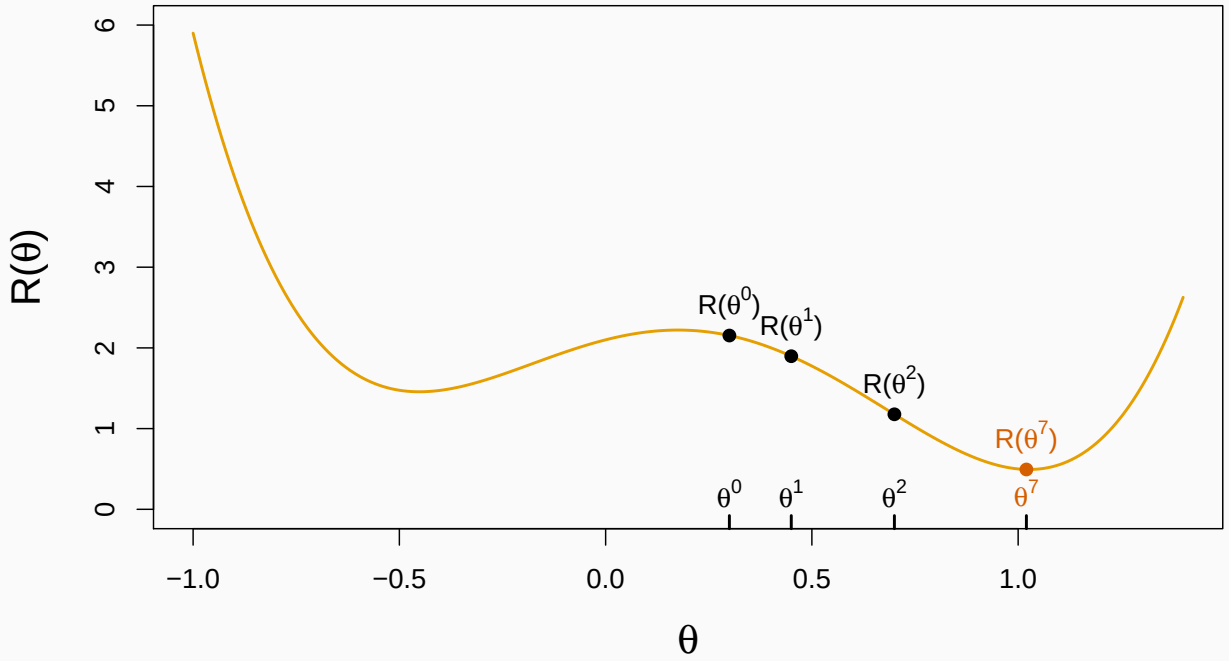
Fitting Neural Networks



$$\min_{\{w_k\}_1^K,\,\beta} \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2, \quad \text{where} \quad f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k\, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\right).$$

Non-Convex Functions and Gradient Descent

Minimize $R(\theta) = \frac{1}{2}\sum_i (y_i - f_\theta(x_i))^2$, where $\theta = (\{w_k\}_1^K, \beta)$



1. Initialize parameters θ_0
2. iterate updates until $R(\theta)$ stops decreasing.
   - Each update: $\theta_{t+1} = \theta_t + \delta$ where δ decreases $R(\theta)$.

Gradient Descent

- Goal: find parameter updates that move downhill in the loss surface.
- Compute the *gradient vector*

$$\nabla R(\theta_t) = \frac{\partial R(\theta)}{\partial \theta}\Big|_{\theta=\theta_t}.$$

- Update rule:

$$\theta_{t+1} = \theta_t - \rho \nabla R(\theta_t),$$

  where ρ is the *learning rate* (e.g., $\rho = 0.001$).

- Gradient points uphill — moving in the opposite direction decreases $R(\theta)$.
- Optimization is hard due to possible **local minima**.

Gradients and Backpropagation

$R(\theta) = \sum_{i=1}^{n} R_i(\theta)$ is a sum, so gradient is sum of gradients.

$$R_i(\theta) = \tfrac{1}{2}(y_i - f_\theta(x_i))^2 = \tfrac{1}{2}\left(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}\right)\right)^2.$$

For ease of notation, let $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$.

Backpropagation uses the *chain rule for differentiation*:

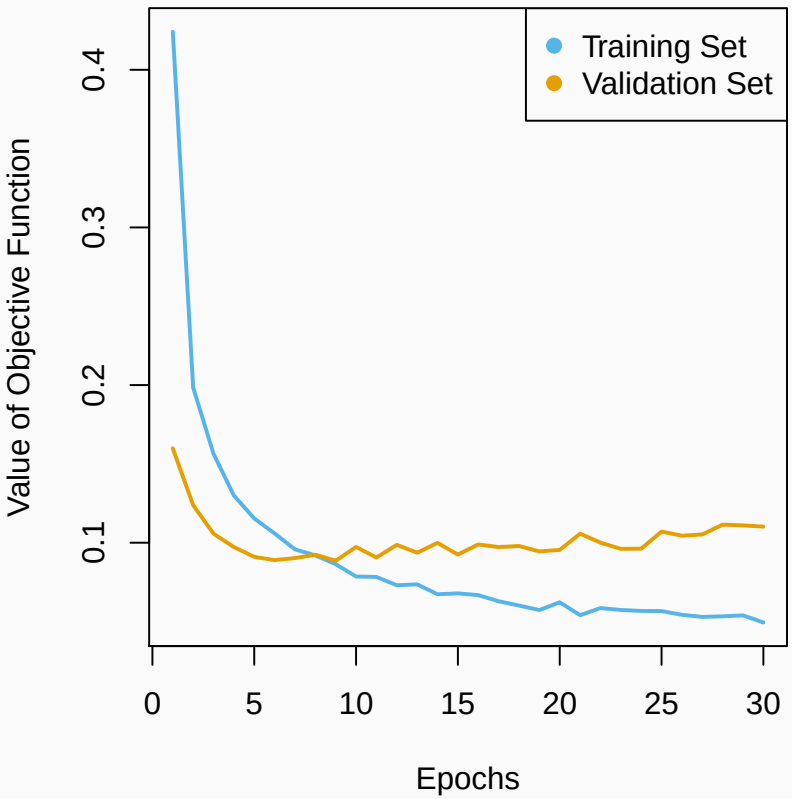
$$\begin{aligned}
\frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} \\
&= -(y_i - f_\theta(x_i)) \cdot g(z_{ik}),
\end{aligned}$$

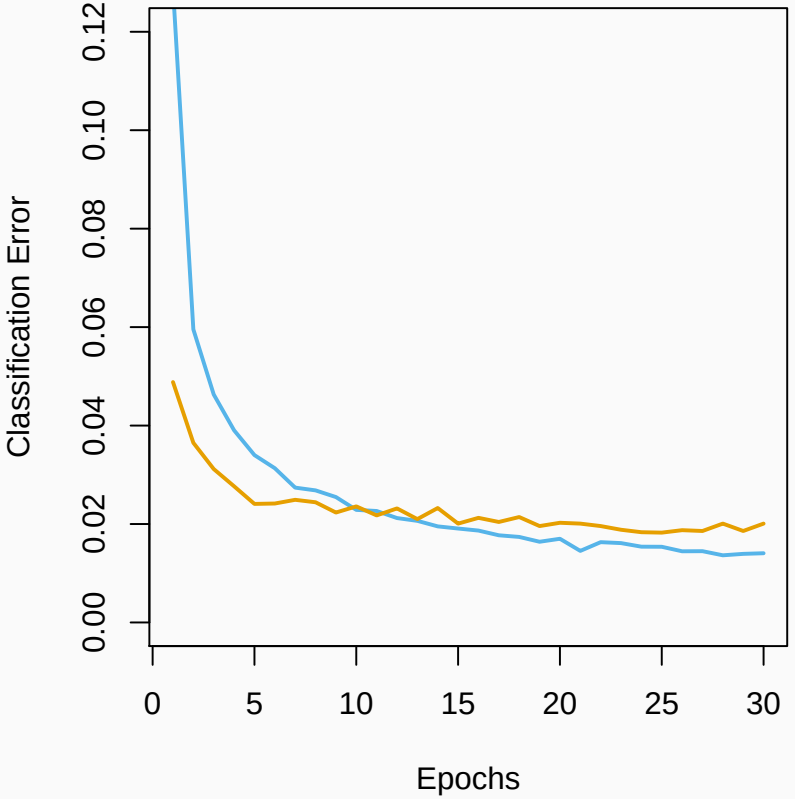
$$\begin{aligned}
\frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \\
&= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij}.
\end{aligned}$$

Training Tricks

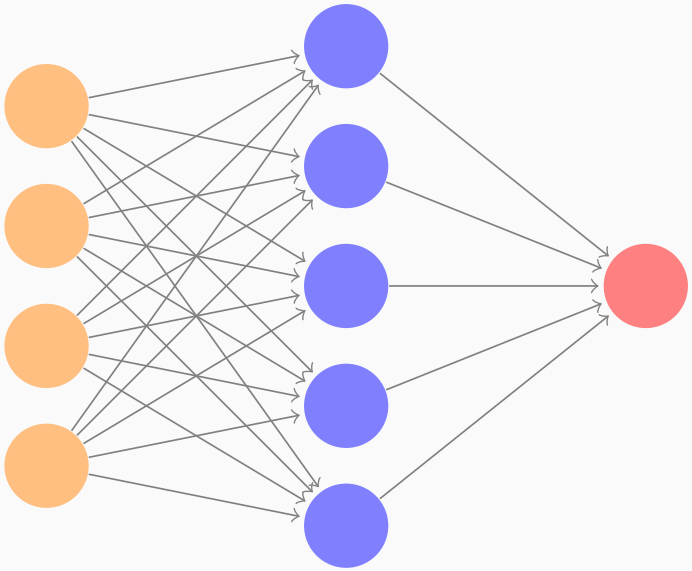
- *Slow learning:* small learning rate ρ slows convergence; *early stopping* acts as regularization.
- *Stochastic Gradient Descent (SGD):* parameters are updated using small random *minibatches* (e.g., 128 images from MNIST). One full pass over the dataset is called an *epoch* (n/batch size updates per epoch).
- *Regularization:* ridge or lasso can shrink weights at each layer.
- Other forms of regularization: *dropout* and *data augmentation*.

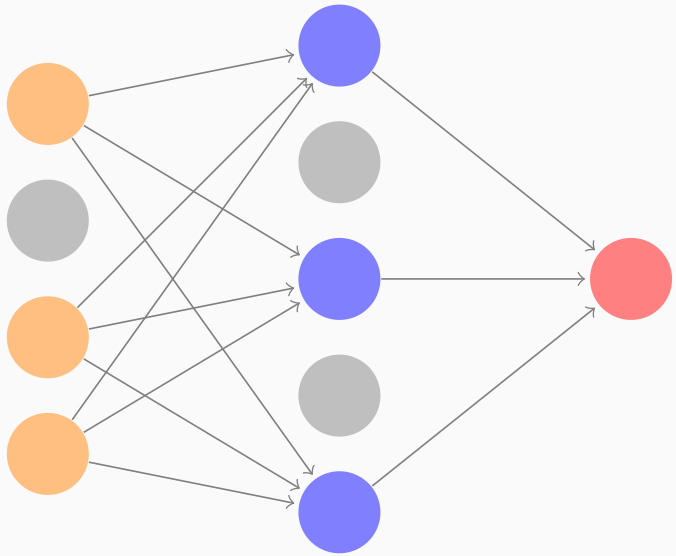
SGD: MNIST example





Dropout Learning





- At each SGD update, randomly remove units with probability ϕ.
- Scale up remaining weights by $1/(1-\phi)$ to maintain total signal.