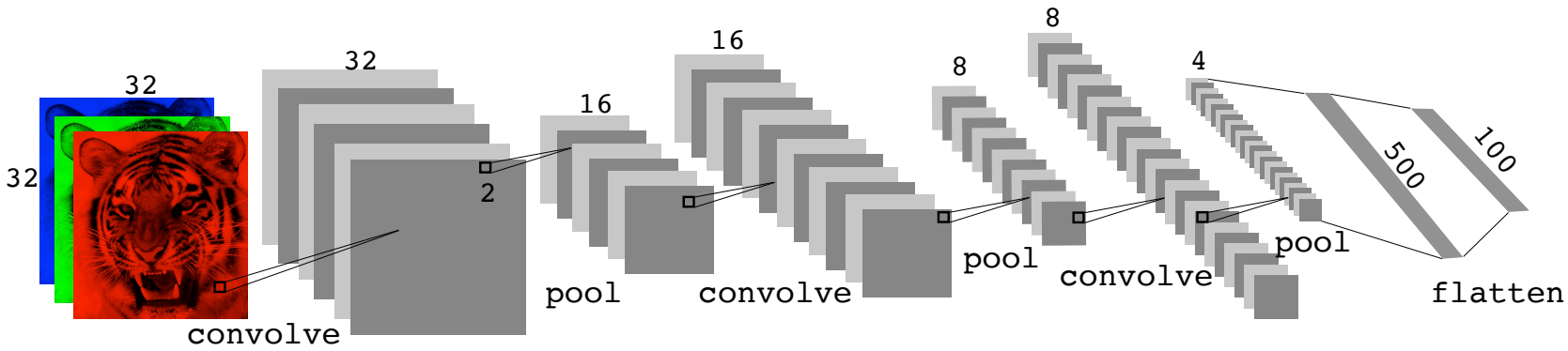
上节课回顾

Architecture of a CNN



- Many convolve + pool layers.
- Filters are typically small, e.g. each channel 3×3.
- Each filter creates a new channel in convolution layer.
- As pooling reduces size, the number of filters/channels is typically increased.
- Number of layers can be very large. E.g. `resnet50` trained on `imagenet` 1000-class image data base has 50 layers!

上节课回顾

1. Prepare training and test data

[49]:
```
(cifar_train,
 cifar_test) = [CIFAR100(root="data",
                         train=train,
                         download=True)
                for train in [True, False]]
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[50]:
```
transform = ToTensor()
cifar_train_X = torch.stack([transform(x) for x in
                             cifar_train.data])
cifar_test_X = torch.stack([transform(x) for x in
                            cifar_test.data])
cifar_train = TensorDataset(cifar_train_X,
                            torch.tensor(cifar_train.targets))
cifar_test = TensorDataset(cifar_test_X,
                           torch.tensor(cifar_test.targets))
```

The `CIFAR100` dataset consists of 50,000 training images, each represented b
standardize as we did for the digits, but keep the array structure. This is accomp

Creating the data module is similar to the `MNIST` example.

[51]:
```
cifar_dm = SimpleDataModule(cifar_train,
                            cifar_test,
                            validation=0.2,
                            num_workers=max_num_workers,
                            batch_size=128)
```

上节课回顾

2. Specify CNN architecture

[54]:
```
class BuildingBlock(nn.Module):

    def __init__(self,
                 in_channels,
                 out_channels):

        super(BuildingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=(3,3),
                              padding='same')
        self.activation = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=(2,2))

    def forward(self, x):
        return self.pool(self.activation(self.conv(x)))
```

Notice that we used the `padding = "same"` argument to `nn.Conv2d()`, w
channels in the input layer. We use a 3×3 convolution filter for each channel i

In forming our deep learning model for the `CIFAR100` data, we use several of
can be combined in other modules. Ultimately, everything is fit by a generic trai

[55]:
```
class CIFARModel(nn.Module):

    def __init__(self):
        super(CIFARModel, self).__init__()
        sizes = [(3,32),
                 (32,64),
                 (64,128),
                 (128,256)]
        self.conv = nn.Sequential(*[BuildingBlock(in_, out_)
                                    for in_, out_ in sizes])

        self.output = nn.Sequential(nn.Dropout(0.5),
                                    nn.Linear(2*2*256, 512),
                                    nn.ReLU(),
                                    nn.Linear(512, 100))
    def forward(self, x):
        val = self.conv(x)
        val = torch.flatten(val, start_dim=1)
        return self.output(val)
```

上节课回顾

3. Fit the parameters in CNN

[57]:
```
cifar_optimizer = RMSprop(cifar_model.parameters(), lr=0.001)
cifar_module = SimpleModule.classification(cifar_model,
                                           num_classes=100,
                                           optimizer=cifar_optimizer)
cifar_logger = CSVLogger('logs', name='CIFAR100')
```

[58]:
```
cifar_trainer = Trainer(deterministic=True,
                        max_epochs=30,
                        logger=cifar_logger,
                        enable_progress_bar=False,
                        callbacks=[ErrorTracker()])
cifar_trainer.fit(cifar_module,
                  datamodule=cifar_dm)
```

上节课回顾

Homework 2 (deadline: Oct. 31)

The Goal of this Homework:

Use Convolutional Neural Network (CNN) and several meteorological fields to predict monthly precipitation rate over the Beijing region. Specifically, you need to predict the monthly precipitation rate during 2021-2023 over the Beijing region (111E-121E, 35N-45N). Hint: you can improve the prediction skill either through the input meteorological field data or CNN configuration.

Homework Requirement:

1) understand the example python code,
2) do your best to improve the prediction skill by reducing RMSE value in the end of the code,
3) write down your investigation about this problem and explain your ideas why your method would improve the prediction.

Please also note that

1) you can find an example python script and meteorological field data in the homework folder,
2) you can choose to download other meteorological field data from the ERA5 website (monthly/hourly, pressure levels/single level). To download data, you have to first register this website, click "Download" button, Select "Reanalysis", Variable, Year, Month, Geographical area and so on. It is recommended to pick "NetCDF4 (Experimental)" data format so that you can read the data by import netCDF4 in python. Several useful datasets are listed below,
   - https://cds.climate.copernicus.eu/datasets/reanalysis-era5-pressure-levels-monthly-means?tab=overview
   - https://cds.climate.copernicus.eu/datasets/reanalysis-era5-single-levels-monthly-means?tab=overview
   - https://cds.climate.copernicus.eu/datasets/reanalysis-era5-pressure-levels?tab=overview
   - https://cds.climate.copernicus.eu/datasets/reanalysis-era5-single-levels?tab=overview
3) Your training data should not include precipitation data during 2021-2023 over the Beijing region,
4) You should only use the variable mtpr in "mtpr_ERA5.nc" as your target data,
5) Please do not change the seed value, seed_everything(0, workers=True), in the python code,
6) Besides, those who get the best results (the lowest RMSE) in a reasonable way will receive a small gift as a reward.

Document Classification: IMDB Movie Reviews

The `IMDB` corpus consists of user-supplied movie ratings for a large collection of movies. Each has been labeled for `sentiment` as `positive` or `negative`. Here is the beginning of a negative review:

> This has to be one of the worst films of the 1990s. When my friends & I were watching this film (being the target audience it was aimed at) we just sat & watched the first half an hour with our jaws touching the floor at how bad it really was. The rest of the time, everyone else in the theater just started talking to each other, leaving or generally crying into their popcorn ...

We have labeled training and test sets, each consisting of 25,000 reviews, and each balanced with regard to sentiment.

Document Classification: IMDB Movie Reviews

The `IMDB` corpus consists of user-supplied movie ratings for a large collection of movies. Each has been labeled for `sentiment` as `positive` or `negative`. Here is the beginning of a negative review:

> This has to be one of the worst films of the 1990s. When my friends & I were watching this film (being the target audience it was aimed at) we just sat & watched the first half an hour with our jaws touching the floor at how bad it really was. The rest of the time, everyone else in the theater just started talking to each other, leaving or generally crying into their popcorn ...

We have labeled training and test sets, each consisting of 25,000 reviews, and each balanced with regard to sentiment.

We wish to build a classifier to predict the sentiment of a review.

Featurization: Bag-of-Words

Documents have different lengths, and consist of sequences of words. How do we create features X to characterize a document?

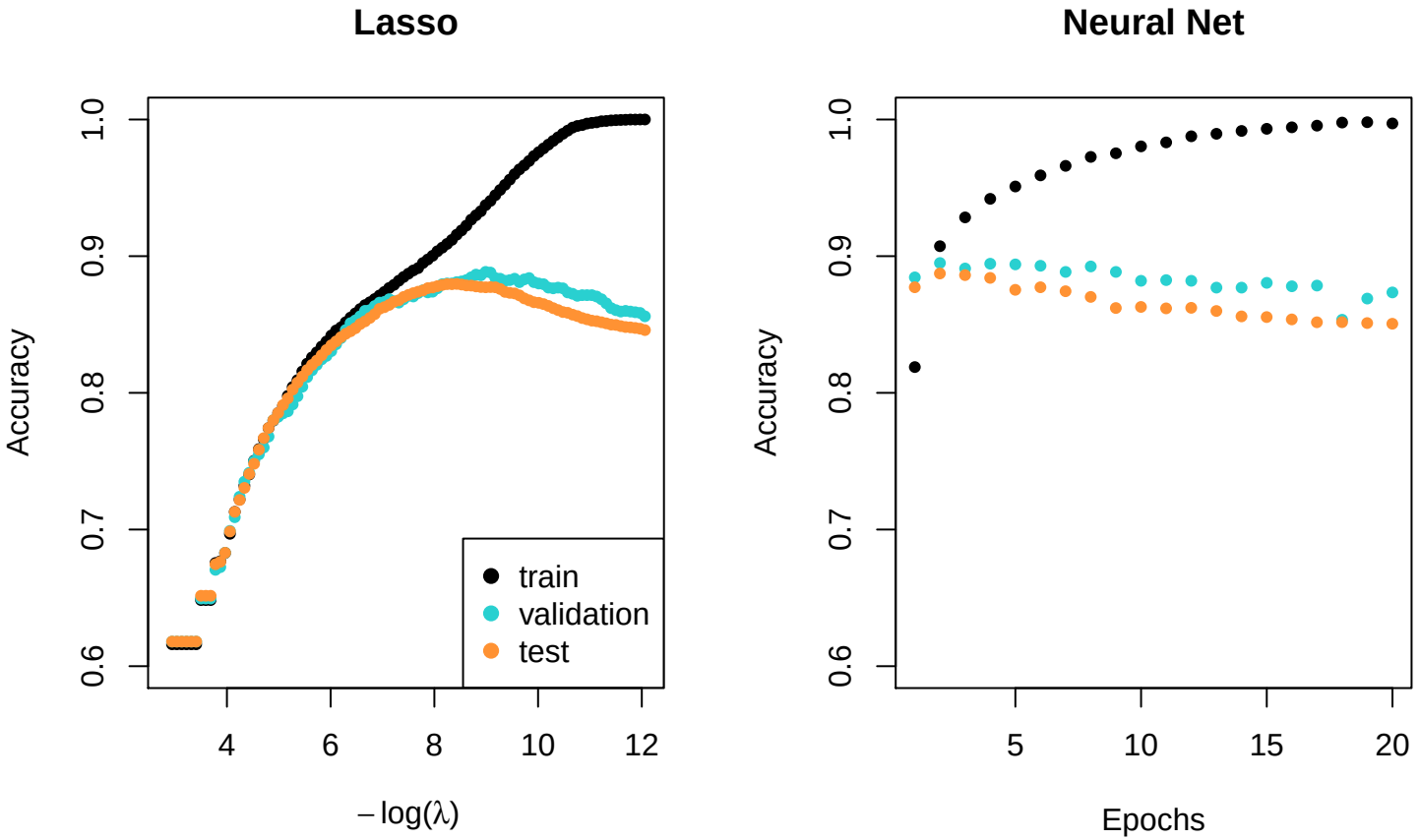
- From a dictionary, identify the $10K$ most frequently occurring words.
- Create a binary vector of length $p = 10K$ for each document, and score a 1 in every position that the corresponding word occurred.
- With n documents, we now have a $n \times p$ *sparse* feature matrix $\mathbf{X}$.
- We compare a lasso logistic regression model to a two-hidden-layer neural network on the next slide. (No convolutions here!)

Featurization: Bag-of-Words

Documents have different lengths, and consist of sequences of words. How do we create features X to characterize a document?

- From a dictionary, identify the $10K$ most frequently occurring words.
- Create a binary vector of length $p = 10K$ for each document, and score a 1 in every position that the corresponding word occurred.
- With n documents, we now have a $n \times p$ *sparse* feature matrix $\mathbf{X}$.
- We compare a lasso logistic regression model to a two-hidden-layer neural network on the next slide. (No convolutions here!)
- Bag-of-words are *unigrams*. We can instead use *bigrams* (occurrences of adjacent word pairs), and in general *m-grams*.

Lasso versus Neural Network — IMDB Reviews



- Simpler lasso logistic regression model works as well as neural network in this case.
- `glmnet` was used to fit the lasso model, and is very effective because it can exploit sparsity in the $\mathbf{X}$ matrix.

Recurrent Neural Networks

Often data arise as sequences:

- Documents are sequences of words, and their relative positions have meaning.
- Time-series such as weather data or financial indices.
- Recorded speech or music.
- Handwriting, such as doctor's notes.

RNNs build models that take into account this sequential nature of the data, and build a memory of the past.

Recurrent Neural Networks

Often data arise as sequences:

- Documents are sequences of words, and their relative positions have meaning.
- Time-series such as weather data or financial indices.
- Recorded speech or music.
- Handwriting, such as doctor's notes.

RNNs build models that take into account this sequential nature of the data, and build a memory of the past.

- The feature for each observation is a *sequence* of vectors $X = \{X_1, X_2, \ldots, X_L\}$.

Recurrent Neural Networks

Often data arise as sequences:

- Documents are sequences of words, and their relative positions have meaning.
- Time-series such as weather data or financial indices.
- Recorded speech or music.
- Handwriting, such as doctor's notes.

RNNs build models that take into account this sequential nature of the data, and build a memory of the past.

- The feature for each observation is a *sequence* of vectors $X = \{X_1, X_2, \ldots, X_L\}$.
- The target Y is often of the usual kind — e.g. a single variable such as `Sentiment`, or a one-hot vector for multiclass.

Recurrent Neural Networks

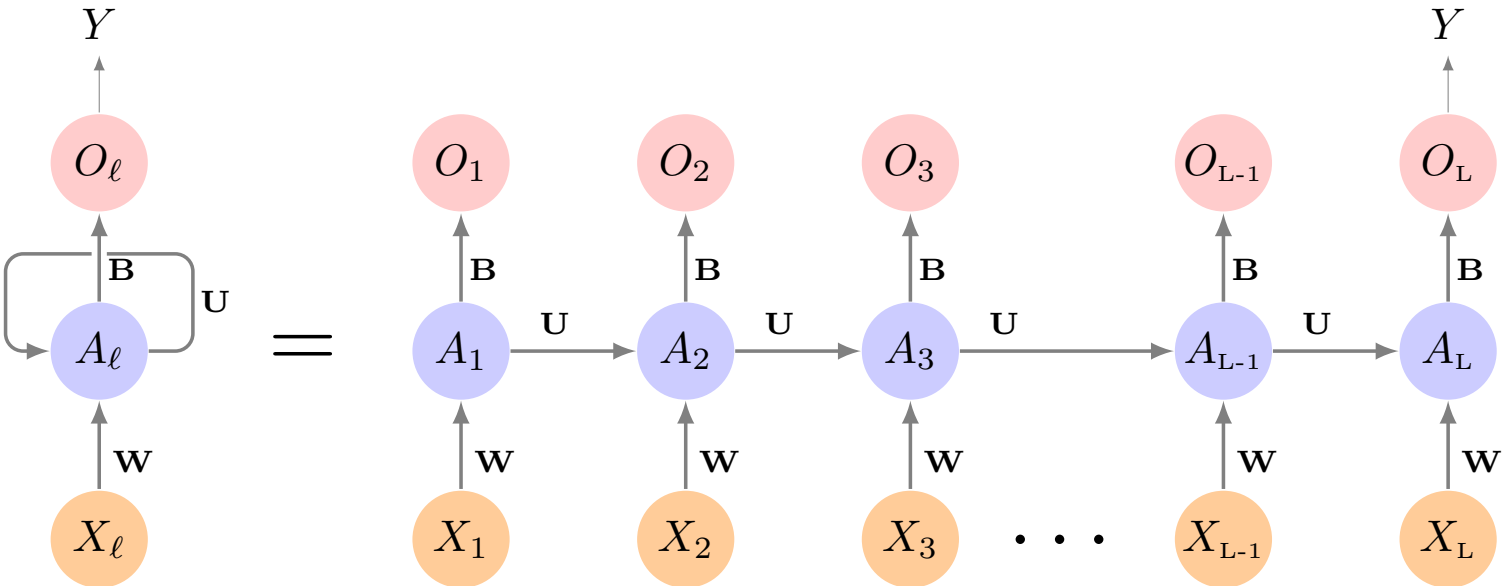
Often data arise as sequences:

- Documents are sequences of words, and their relative positions have meaning.
- Time-series such as weather data or financial indices.
- Recorded speech or music.
- Handwriting, such as doctor's notes.

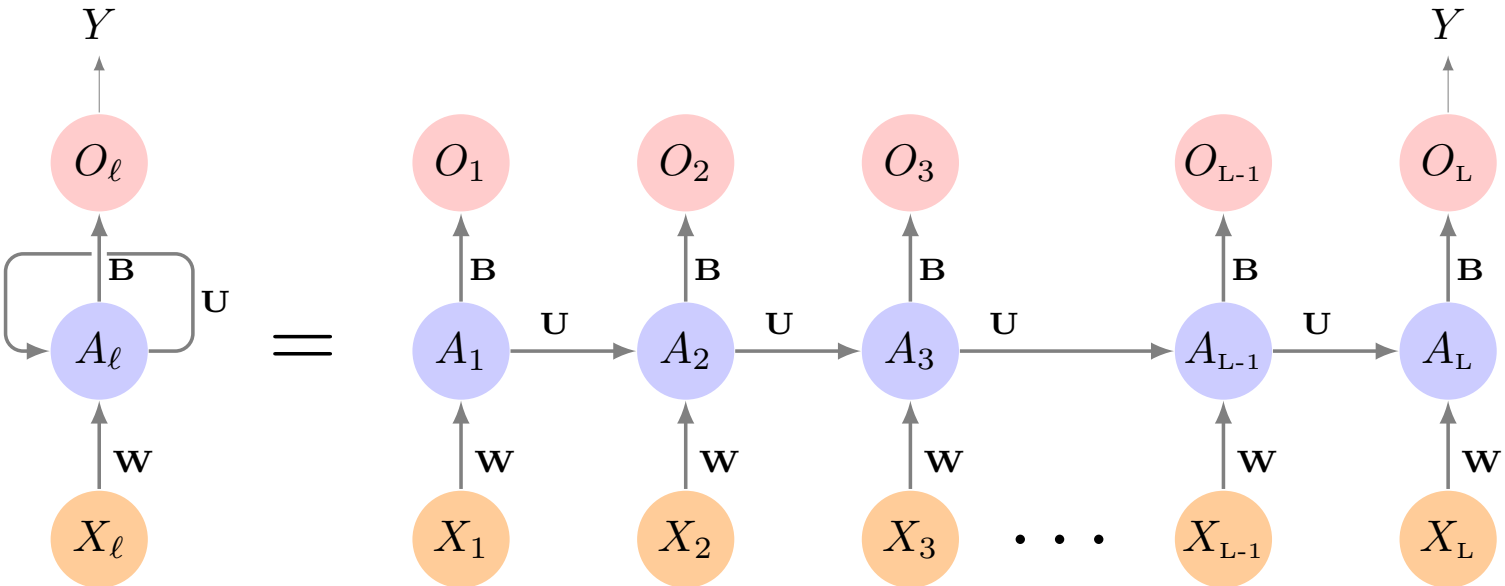
RNNs build models that take into account this sequential nature of the data, and build a memory of the past.

- The feature for each observation is a *sequence* of vectors $X = \{X_1, X_2, \ldots, X_L\}$.
- The target Y is often of the usual kind — e.g. a single variable such as `Sentiment`, or a one-hot vector for multiclass.
- However, Y can also be a sequence, such as the same document in a different language.

Simple Recurrent Neural Network Architecture

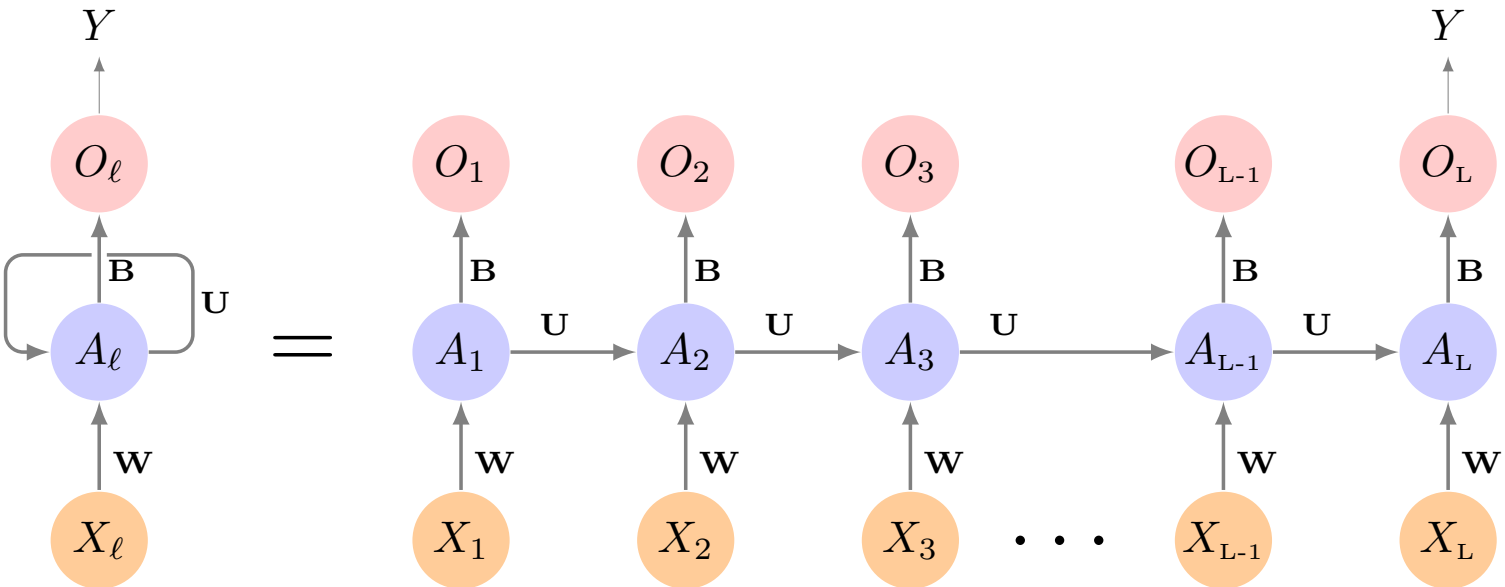


Simple Recurrent Neural Network Architecture



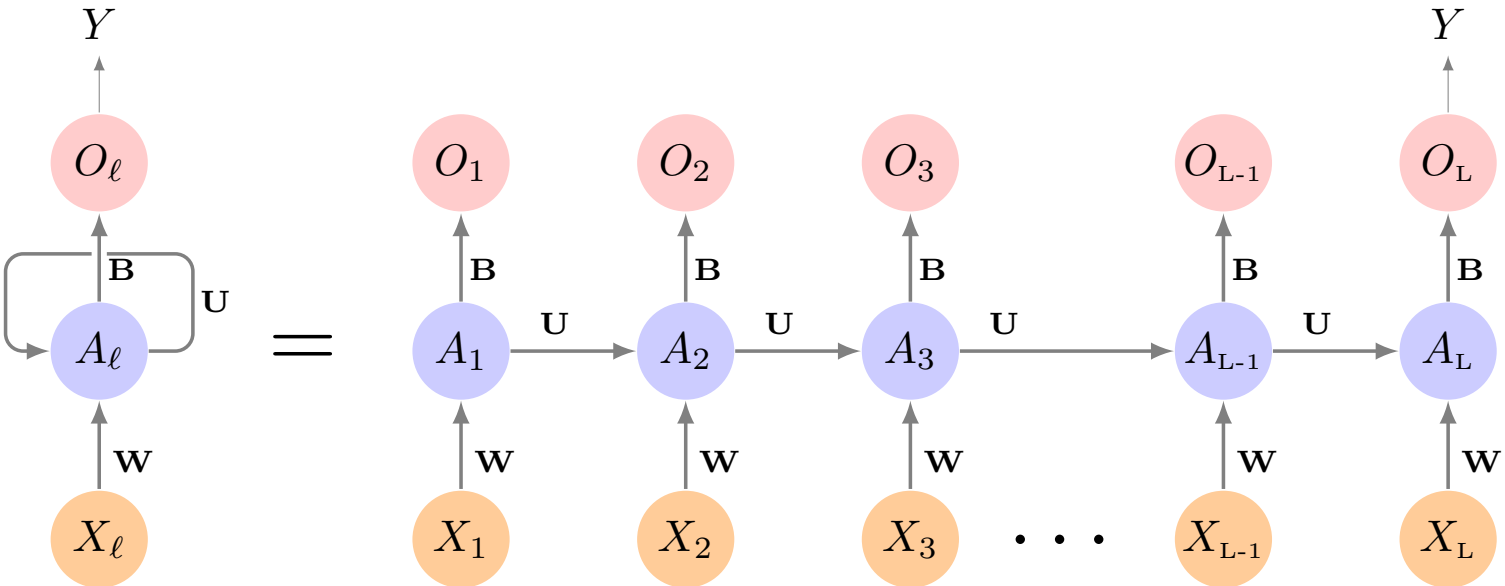
- The hidden layer is a sequence of vectors A_ℓ, receiving as input X_ℓ as well as $A_{\ell-1}$. A_ℓ produces an output O_ℓ.

Simple Recurrent Neural Network Architecture



- The hidden layer is a sequence of vectors A_ℓ, receiving as input X_ℓ as well as $A_{\ell-1}$. A_ℓ produces an output O_ℓ.
- The *same* weights $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ are used at each step in the sequence — hence the term *recurrent*.

Simple Recurrent Neural Network Architecture



- The hidden layer is a sequence of vectors A_ℓ, receiving as input X_ℓ as well as $A_{\ell-1}$. A_ℓ produces an output O_ℓ.
- The *same* weights $\mathbf{W}$, $\mathbf{U}$ and $\mathbf{B}$ are used at each step in the sequence — hence the term *recurrent*.
- The A_ℓ sequence represents an evolving model for the response that is updated as each element X_ℓ is processed.

RNN in Detail

Suppose $X_\ell = (X_{\ell 1}, X_{\ell 2}, \ldots, X_{\ell p})$ has p components, and $A_\ell = (A_{\ell 1}, A_{\ell 2}, \ldots, A_{\ell K})$ has K components. Then the computation at the kth components of hidden unit A_ℓ is

$$\begin{aligned} A_{\ell k} &= g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{\ell j} + \sum_{s=1}^{K} u_{ks} A_{\ell-1,s}\Big) \\ O_\ell &= \beta_0 + \sum_{k=1}^{K} \beta_k A_{\ell k} \end{aligned}$$

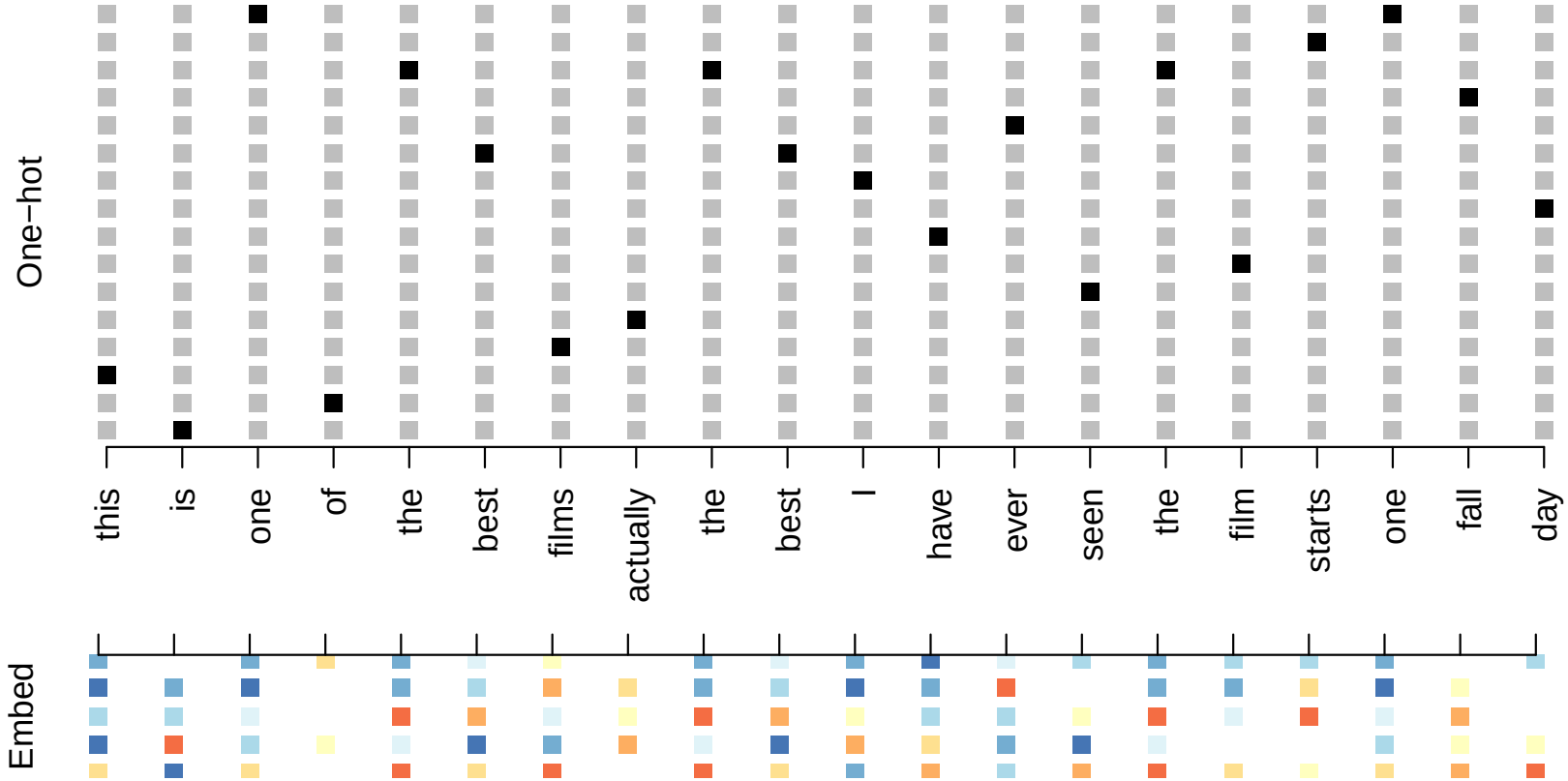
Often we are concerned only with the prediction O_L at the last unit. For squared error loss, and n sequence/response pairs, we would minimize

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

RNN and IMDB Reviews

- The document feature is a sequence of words $\{\mathcal{W}_\ell\}_1^L$. We typically truncate/pad the documents to the same number L of words (we use $L = 500$).
- Each word $\mathcal{W}_\ell$ is represented as a *one-hot encoded* binary vector X_ℓ (dummy variable) of length $10K$, with all zeros and a single one in the position for that word in the dictionary.
- This results in an extremely sparse feature representation, and would not work well.
- Instead we use a lower-dimensional pretrained *word embedding* matrix $\mathbf{E}$ ($m \times 10K$, next slide).
- This reduces the binary feature vector of length $10K$ to a real feature vector of dimension $m \ll 10K$ (e.g. m in the low hundreds.)

Word Embedding



this is one of the best films actually the best I have ever seen the film starts one fall day $\cdots$.

Embeddings are pretrained on very large corpora of documents, using methods similar to principal components. `word2vec` and `GloVe` are popular.

RNN on IMDB Reviews

- After a lot of work, the results are a disappointing 76% accuracy.

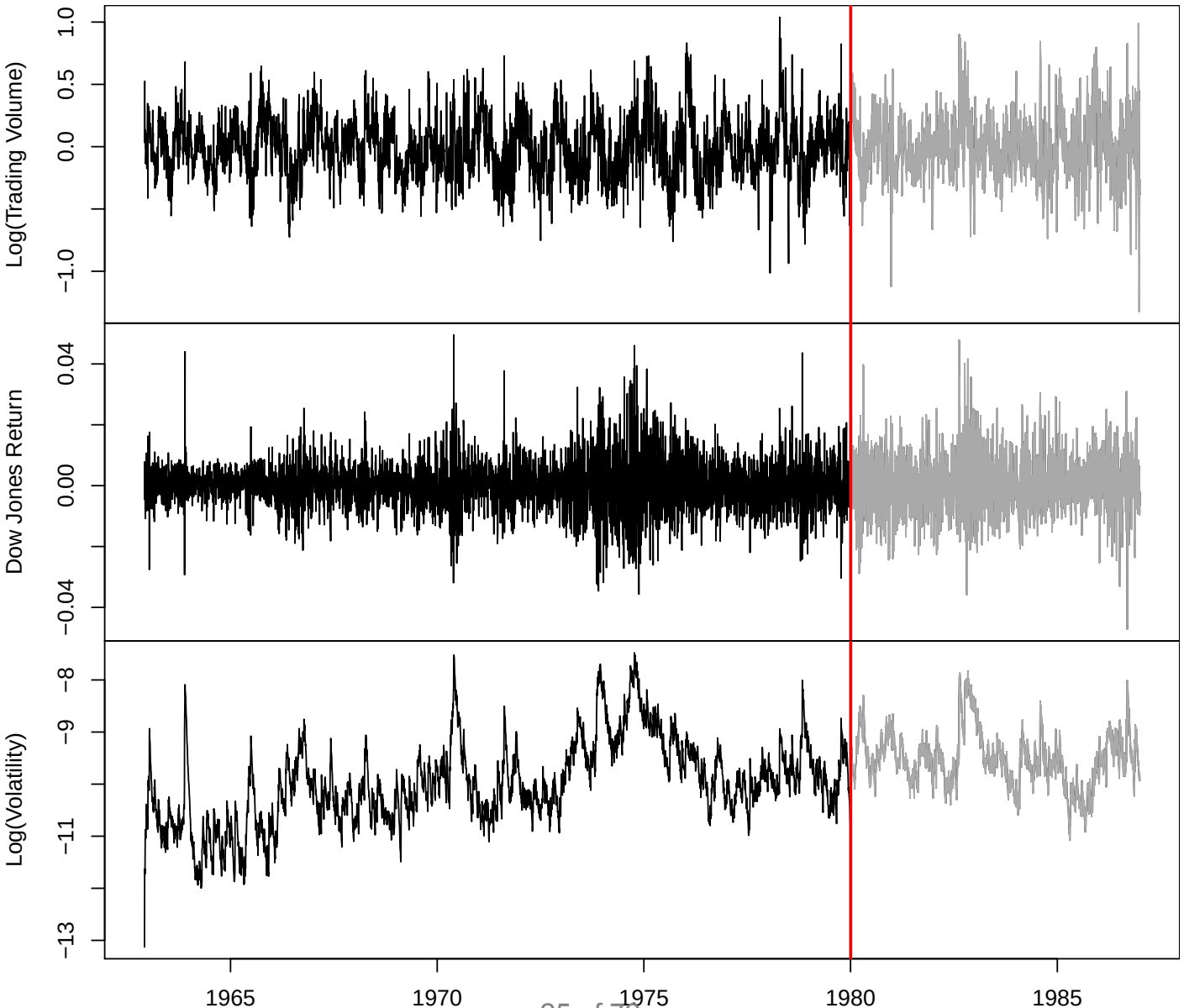
RNN on IMDB Reviews

- After a lot of work, the results are a disappointing 76% accuracy.
- We then fit a more exotic RNN than the one displayed — a *LSTM* with *long and short term memory.* Here 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). Now we get 87% accuracy, slightly less than the 88% achieved by `glmnet`.

RNN on IMDB Reviews

- After a lot of work, the results are a disappointing 76% accuracy.
- We then fit a more exotic RNN than the one displayed — a *LSTM* with *long and short term memory.* Here 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). Now we get 87% accuracy, slightly less than the 88% achieved by `glmnet`.
- These data have been used as a benchmark for new RNN architectures. The best reported result found at the time of writing (2020) was around 95%. We point to a *leaderboard* in Section 10.5.1.

Time Series Forecasting



New-York Stock Exchange Data

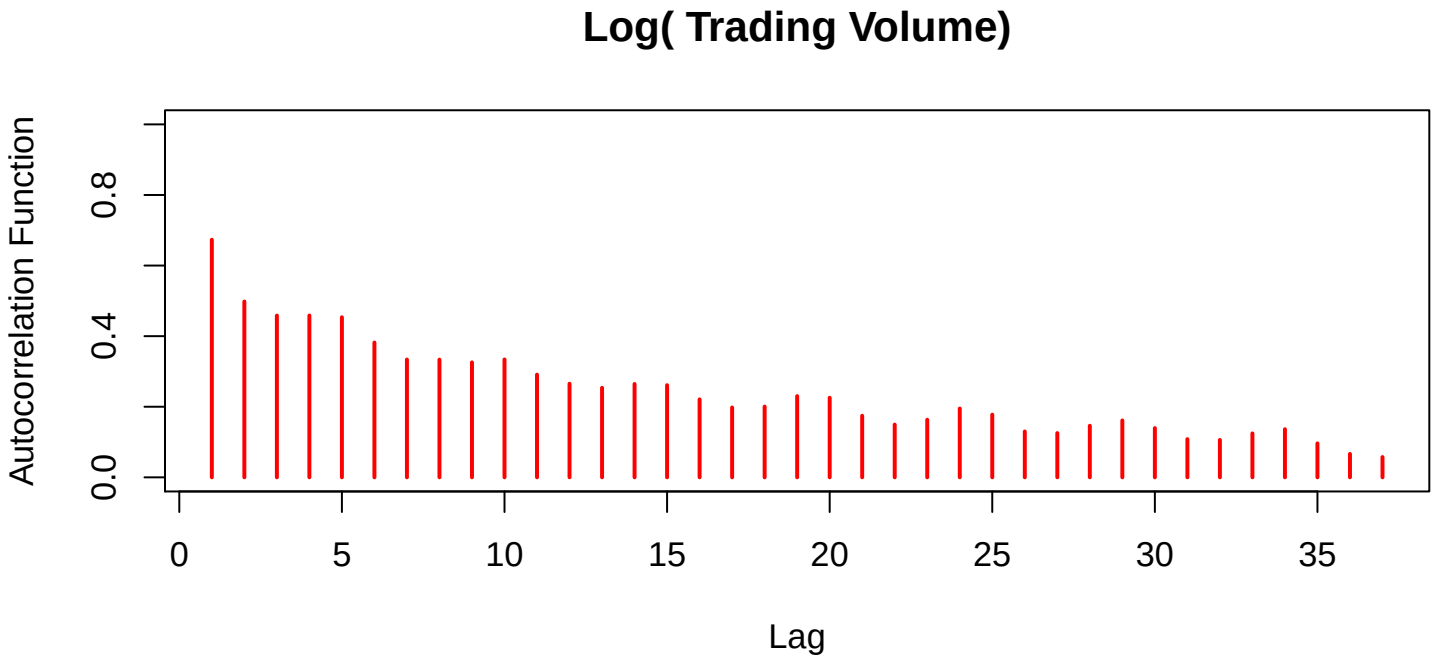
Shown in previous slide are three daily time series for the period December 3, 1962 to December 31, 1986 (6,051 trading days):

- `Log trading volume`. This is the fraction of all outstanding shares that are traded on that day, relative to a 100-day moving average of past turnover, on the log scale.
- `Dow Jones return`. This is the difference between the log of the Dow Jones Industrial Index on consecutive trading days.
- `Log volatility`. This is based on the absolute values of daily price movements.

Goal: predict `Log trading volume` tomorrow, given its observed values up to today, as well as those of `Dow Jones return` and `Log volatility`.

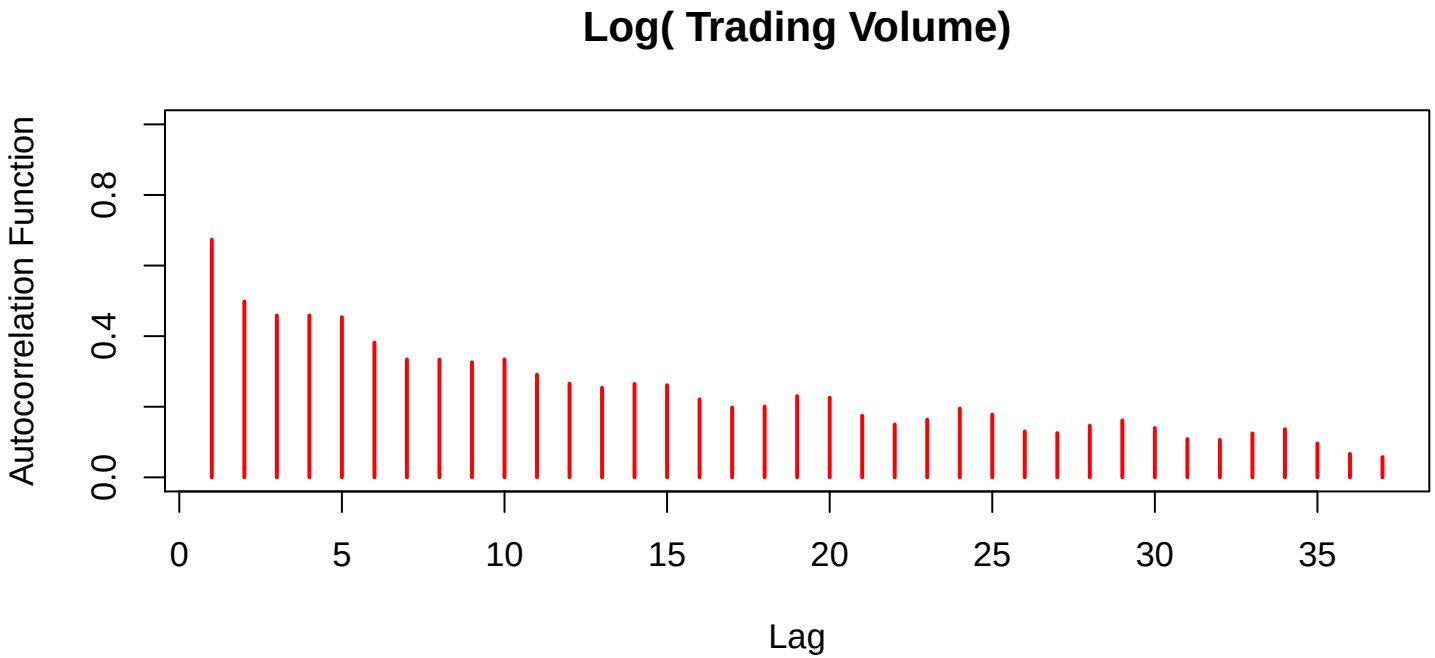
These data were assembled by LeBaron and Weigend (1998) *IEEE Transactions on Neural Networks*, 9(1): 213–220.

Autocorrelation



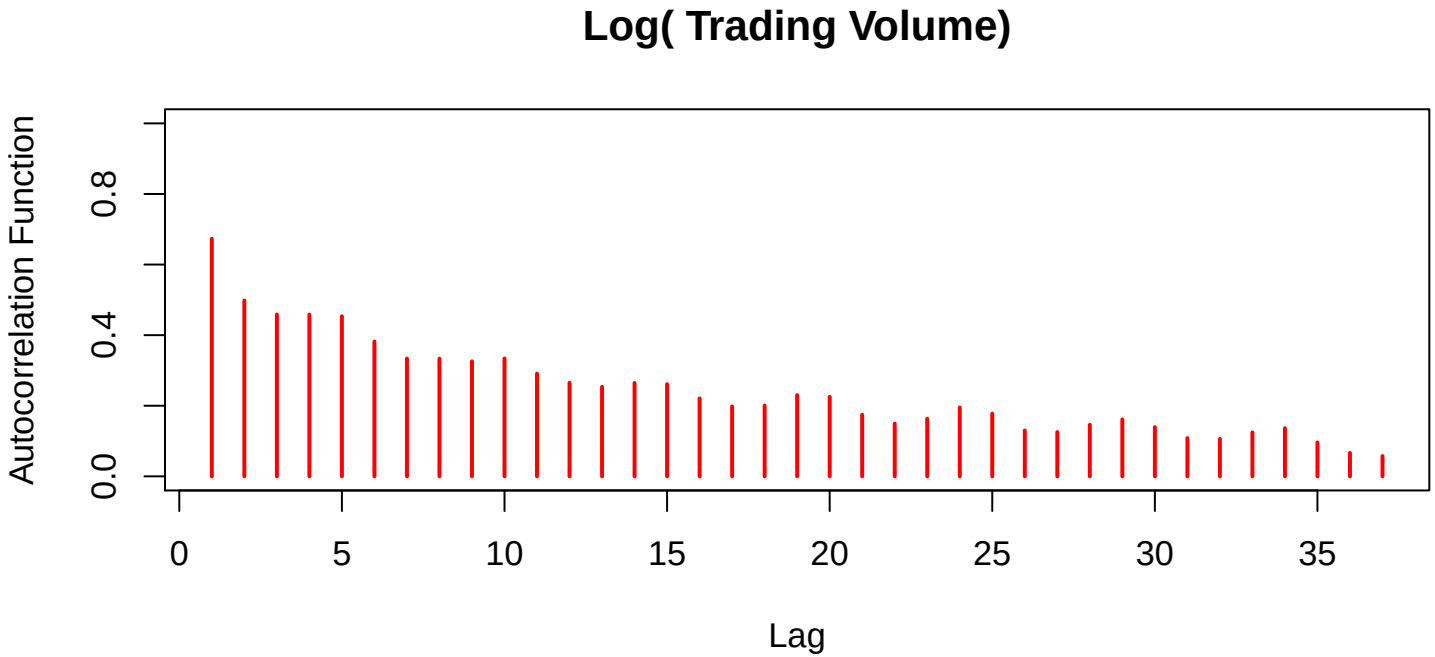
- The *autocorrelation* at lag ℓ is the correlation of all pairs $(v_t, v_{t-\ell})$ that are ℓ trading days apart.

Autocorrelation



- The *autocorrelation* at lag ℓ is the correlation of all pairs $(v_t, v_{t-\ell})$ that are ℓ trading days apart.
- These sizable correlations give us confidence that past values will be helpful in predicting the future.

Autocorrelation



- The *autocorrelation* at lag ℓ is the correlation of all pairs $(v_t, v_{t-\ell})$ that are ℓ trading days apart.
- These sizable correlations give us confidence that past values will be helpful in predicting the future.
- This is a curious prediction problem: the response v_t is also a feature $v_{t-\ell}$!

RNN Forecaster

We only have one series of data! How do we set up for an RNN?

We extract many short mini-series of input sequences $X = \{X_1, X_2, \ldots, X_L\}$ with a predefined length L known as the *lag*:

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

Since $T = 6,051$, with $L = 5$ we can create $6,046$ such (X, Y) pairs.
We use the first $4,281$ as training data, and the following $1,770$ as test data. We fit an RNN with 12 hidden units per lag step (i.e. per A_ℓ.)

RNN Results for NYSE Data

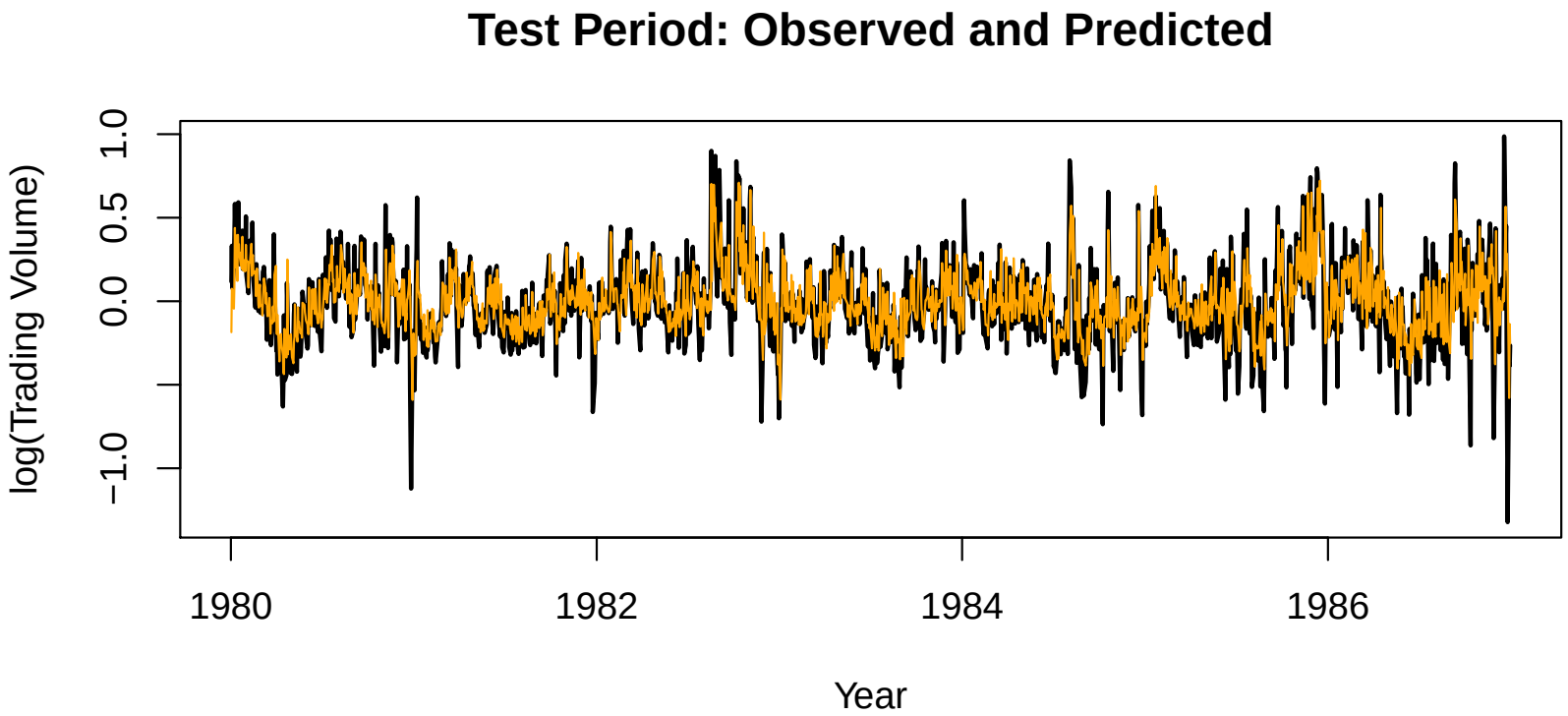


Figure shows predictions and truth for test period.

$R^2 = 0.42$ for RNN
$R^2 = 0.18$ for *straw man* — use yesterday's value of `Log trading volume` to predict that of today.

Autoregression Forecaster

The RNN forecaster is similar in structure to a traditional *autoregression* procedure.

$$\mathbf{y} = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ v_{L+3} \\ \vdots \\ v_T \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}.$$

Fit an OLS regression of $\mathbf{y}$ on $\mathbf{M}$, giving

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

Known as an *order-L autoregression* model or AR(L).
For the `NYSE` data we can include lagged versions of `DJ_return` and `log_volatility` in matrix $\mathbf{M}$, resulting in $3L + 1$ columns.

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) model fit by neural network.

$R^2 = 0.46$ for all models if we include `day_of_week` of day being predicted.

Summary of RNNs

- We have presented the simplest of RNNs. Many more complex variations exist.
- One variation treats the sequence as a one-dimensional image, and uses CNNs for fitting. For example, a sequence of words using an embedding representation can be viewed as an image, and the CNN convolves by sliding a convolutional filter along the sequence.
- Can have additional hidden layers, where each hidden layer is a sequence, and treats the previous hidden layer as an input sequence.
- Can have output also be a sequence, and input and output share the hidden units. So called `seq2seq` learning are used for language translation.

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

- Often the big successes occur when the *signal to noise ratio* is high — e.g. image recognition and language translation. Datasets are large, and overfitting is not a big problem.

When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

- Often the big successes occur when the *signal to noise ratio* is high — e.g. image recognition and language translation. Datasets are large, and overfitting is not a big problem.
- For noisier data, simpler models can often work better.
  - On the `NYSE` data, the AR(5) model is much simpler than a RNN, and performed as well.
  - On the `IMDB` review data, the linear model fit by `glmnet` did as well as the neural network, and better than the RNN.

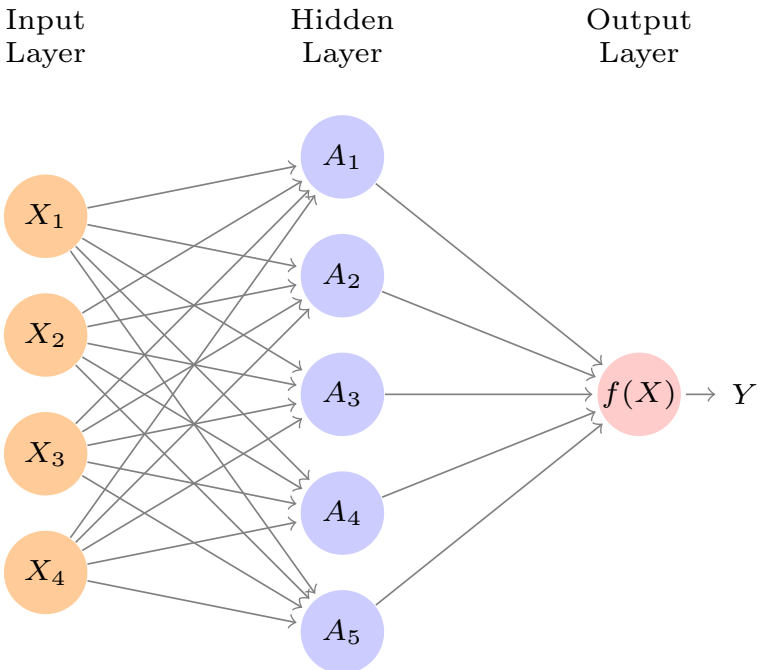
When to Use Deep Learning

- CNNs have had enormous successes in image classification and modeling, and are starting to be used in medical diagnosis. Examples include digital mammography, ophthalmology, MRI scans, and digital X-rays.
- RNNs have had big wins in speech modeling, language translation, and forecasting.

Should we always use deep learning models?

- Often the big successes occur when the *signal to noise ratio* is high — e.g. image recognition and language translation. Datasets are large, and overfitting is not a big problem.
- For noisier data, simpler models can often work better.
  - On the `NYSE` data, the AR(5) model is much simpler than a RNN, and performed as well.
  - On the `IMDB` review data, the linear model fit by `glmnet` did as well as the neural network, and better than the RNN.
- We endorse the *Occam's razor* principal — we prefer simpler models if they work as well. More interpretable!

Fitting Neural Networks

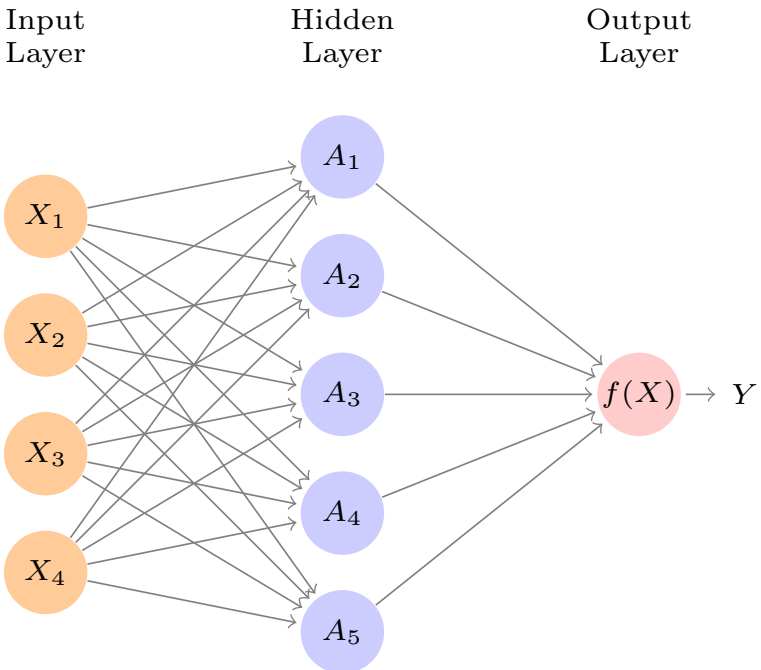


$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

Fitting Neural Networks



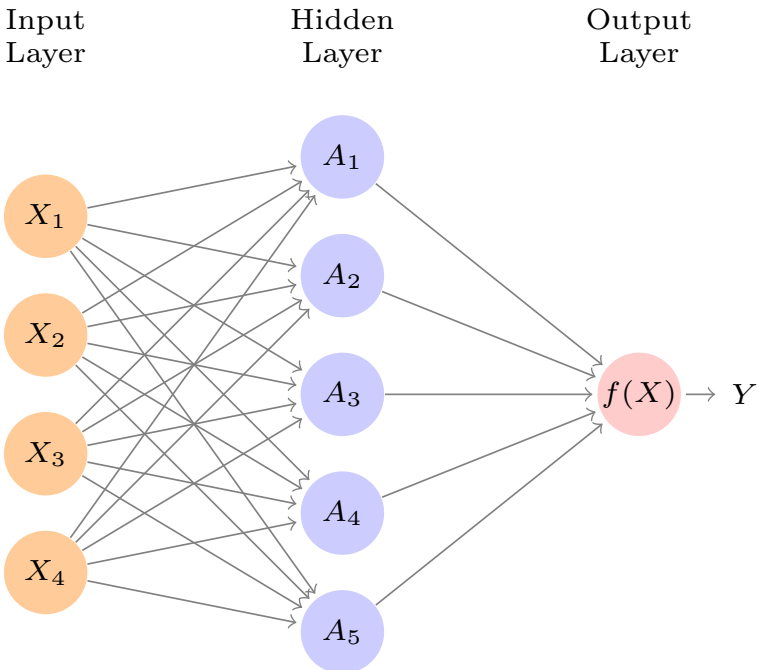
$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

This problem is difficult because the objective is *non-convex*.

Fitting Neural Networks



$$\underset{\{w_k\}_1^K,\,\beta}{\text{minimize}}\ \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where

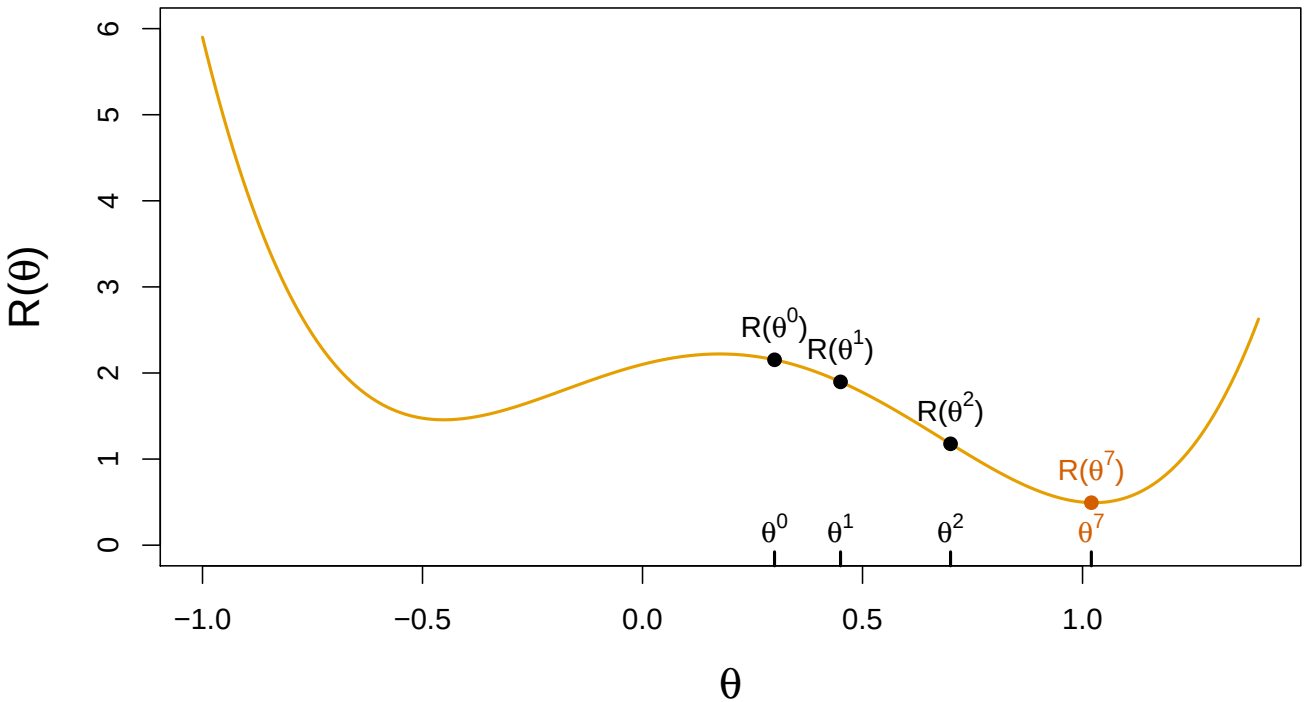
$$f(x_i) = \beta_0 + \sum_{k=1}^{K}\beta_k g\Big(w_{k0} + \sum_{j=1}^{p} w_{kj}x_{ij}\Big).$$

This problem is difficult because the objective is *non-convex*.

Despite this, effective algorithms have evolved that can optimize complex neural network problems efficiently.

Non Convex Functions and Gradient Descent

Let $R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$ with $\theta = (\{w_k\}_1^K,\ \beta)$.



1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$.
2. Iterate until the objective $R(\theta)$ fails to decrease:
   (a) Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ *reduces* the objective; i.e. $R(\theta^{t+1}) < R(\theta^t)$.
   (b) Set $t \leftarrow t + 1$.

Gradient Descent Continued

- In this simple example we reached the *global minimum*.
- If we had started a little to the left of θ^0 we would have gone in the other direction, and ended up in a *local minimum*.
- Although θ is multi-dimensional, we have depicted the process as one-dimensional. It is much harder to identify whether one is in a local minimum in high dimensions.

How to find a direction δ that points downhill? We compute the *gradient vector*

$$\nabla R(\theta^t) = \frac{\partial R(\theta)}{\partial \theta}\Big|_{\theta=\theta^t}$$

i.e. the vector of *partial derivatives* at the current guess θ^t. The gradient points uphill, so our update is $\delta = -\rho \nabla R(\theta^t)$ or

$$\theta^{t+1} \leftarrow \theta^t - \rho \nabla R(\theta^t),$$

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

Gradients and Backpropagation

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

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

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

Backpropagation uses the *chain rule for differentiation*:

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

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.
- An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $60K/128 \approx 469$ for `MNIST`.

Tricks of the Trade

- *Slow learning*. Gradient descent is slow, and a small learning rate ρ slows it even further. With *early stopping*, this is a form of regularization.
- *Stochastic gradient descent*. Rather than compute the gradient using *all* the data, use a small *minibatch* drawn at random at each step. E.g. for `MNIST` data, with $n = 60K$, we use minibatches of 128 observations.
- An *epoch* is a count of iterations and amounts to the number of minibatch updates such that n samples in total have been processed; i.e. $60K/128 \approx 469$ for `MNIST`.
- *Regularization*. Ridge and lasso regularization can be used to shrink the weights at each layer. Two other popular forms of regularization are *dropout* and *augmentation*, discussed next.

Dropout Learning

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.

Dropout Learning

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.

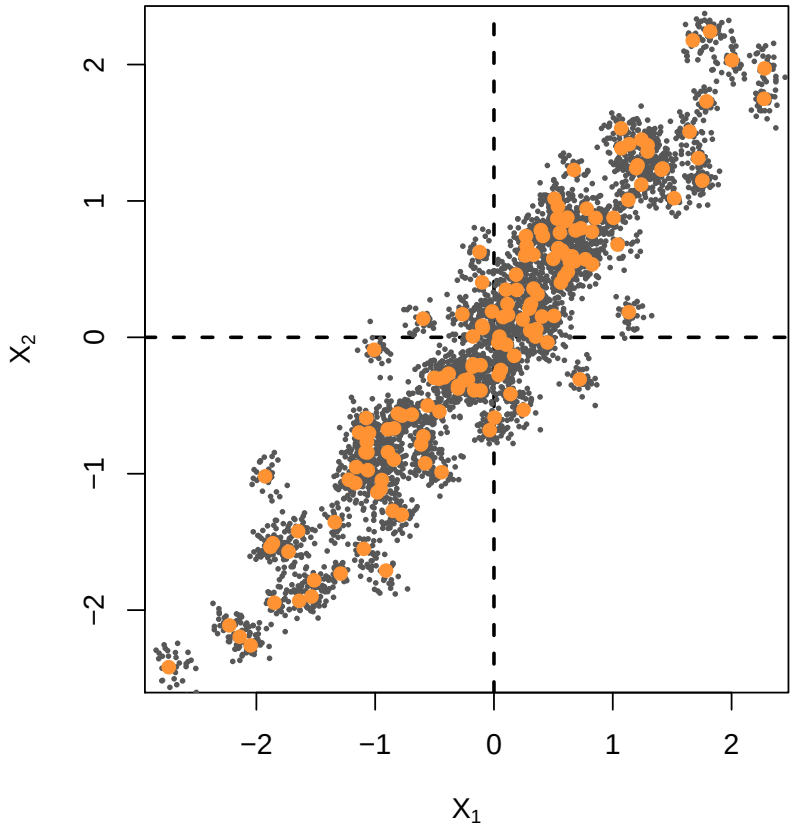
Dropout Learning

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.
- As in ridge, the other units *stand in* for those temporarily removed, and their weights are drawn closer together.

Dropout Learning

- At each SGD update, randomly remove units with probability ϕ, and scale up the weights of those retained by $1/(1-\phi)$ to compensate.
- In simple scenarios like linear regression, a version of this process can be shown to be equivalent to ridge regularization.
- As in ridge, the other units *stand in* for those temporarily removed, and their weights are drawn closer together.
- Similar to randomly omitting variables when growing trees in random forests (Chapter 8).

Ridge and Data Augmentation



- Make many copies of each (x_i, y_i) and add a small amount of Gaussian noise to the x_i — a little cloud around each observation — but *leave the copies of y_i alone!*
- This makes the fit robust to small perturbations in x_i, and is equivalent to ridge regularization in an OLS setting.

Data Augmentation on the Fly

- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.

Data Augmentation on the Fly

- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.

Data Augmentation on the Fly

- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.
- The label is left unchanged — in each case still `tiger`.

Data Augmentation on the Fly

- Data augmentation is especially effective with SGD, here demonstrated for a CNN and image classification.
- Natural transformations are made of each training image when it is sampled by SGD, thus ultimately making a cloud of images around each original training image.
- The label is left unchanged — in each case still `tiger`.
- Improves performance of CNN and is similar to ridge.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.
- Increasing the number of units or layers and again training till zero error sometimes gives *even better* out-of-sample error.

Double Descent

- With neural networks, it seems better to have too many hidden units than too few.
- Likewise more hidden layers better than few.
- Running stochastic gradient descent till zero training error often gives good out-of-sample error.
- Increasing the number of units or layers and again training till zero error sometimes gives *even better* out-of-sample error.

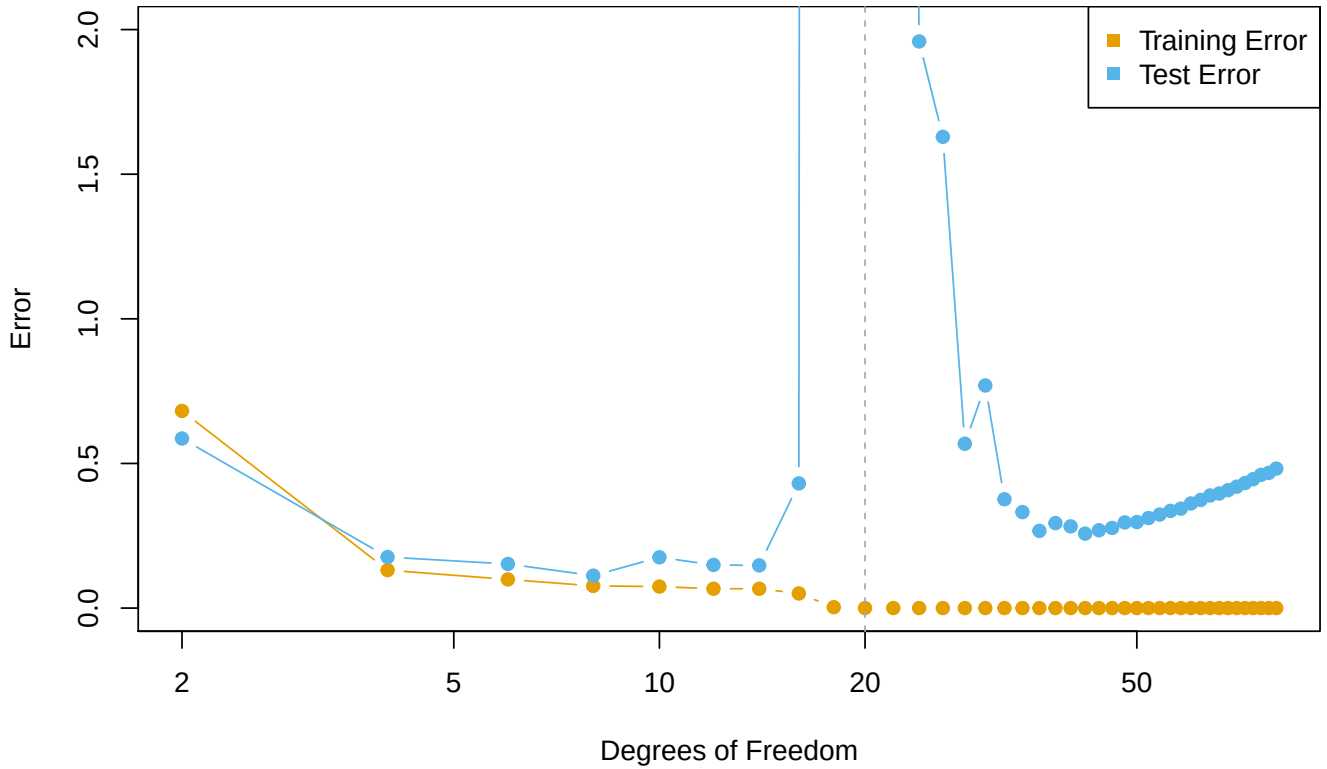
What happened to overfitting and the usual bias-variance trade-off?

Belkin, Hsu, Ma and Mandal (arXiv 2018) *Reconciling Modern Machine Learning and the Bias-Variance Trade-off.*

Simulation

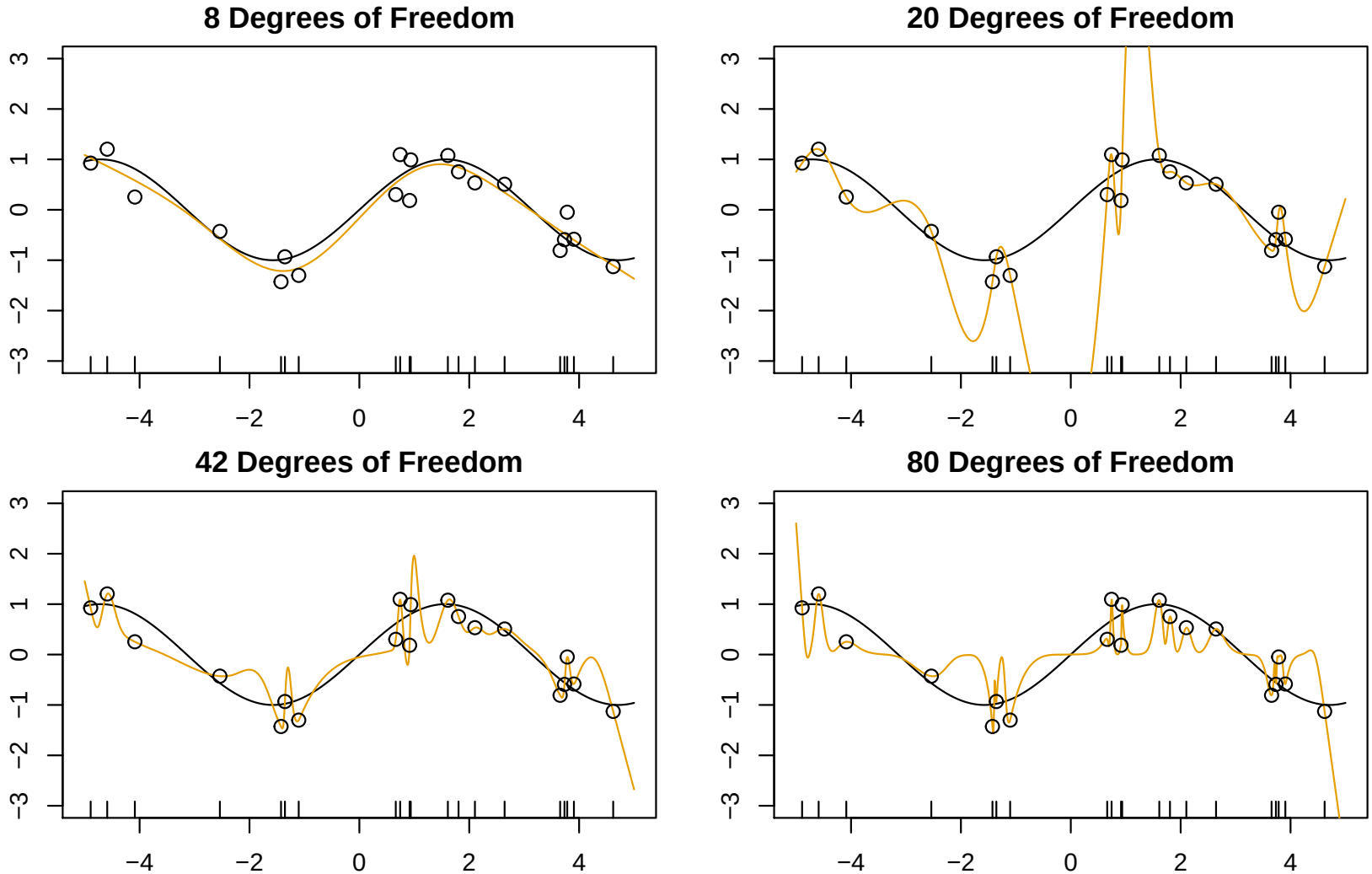
- $y = \sin(x) + \varepsilon$ with $x \sim U[-5, 5]$ and ε Gaussian with S.D. $= 0.3$.
- Training set $n = 20$, test set very large (10K).
- We fit a natural spline to the data (Section 7.4) with d degrees of freedom — i.e. a linear regression onto d basis functions: $\hat{y}_i = \hat{\beta}_1 N_1(x_i) + \hat{\beta}_2 N_2(x_i) + \cdots + \hat{\beta}_d N_d(x_i)$.
- When $d = 20$ we fit the training data exactly, and get all residuals equal to zero.
- When $d > 20$, we still fit the data exactly, but the solution is not unique. Among the zero-residual solutions, we pick the one with *minimum norm* — i.e. the zero-residual solution with smallest $\sum_{j=1}^{d} \hat{\beta}_j^2$.

The Double-Descent Error Curve



- When $d \leq 20$, model is OLS, and we see usual bias-variance trade-off
- When $d > 20$, we revert to minimum-norm. As d increases above 20, $\sum_{j=1}^{d} \hat{\beta}_j^2$ *decreases* since it is easier to achieve zero error, and hence less wiggly solutions.

Less Wiggly Solutions



To achieve a zero-residual solution with $d = 20$ is a real stretch! Easier for larger d.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.
- By analogy, deep and wide neural networks fit by SGD down to zero training error often give good solutions that generalize well.

Some Facts

- In a wide linear model ($p \gg n$) fit by least squares, SGD with a small step size leads to a *minimum norm* zero-residual solution.
- Stochastic gradient *flow* — i.e. the entire path of SGD solutions — is somewhat similar to ridge path.
- By analogy, deep and wide neural networks fit by SGD down to zero training error often give good solutions that generalize well.
- In particular cases with *high signal-to-noise ratio* — e.g. image recognition — are less prone to overfitting; the zero-error solution is mostly signal!

Software

- Wonderful software available for neural networks and deep learning. `Tensorflow` from Google and `PyTorch` from Facebook. Both are `Python` packages.
- In the Chapter 10 lab we demonstrate `tensorflow` and `keras` packages in `R`, which interface to Python. See textbook and online resources for `Rmarkdown` and `Jupyter` notebooks for these and all labs for the second edition of ISLR book.
- The `torch` package in R is available as well, and implements the `PyTorch` dialect. The Chapter 10 lab will be available in this dialect as well; watch the resources page at `www.statlearning.com`.