07. Deep Learning

Table of Contents

Introduction to Deep Learning

- Deep learning is a very active area of research in the machine learning and artificial intelligence communities.
- The cornerstone of deep learning is the neural network.
- Neural networks rose to fame in the late 1980s. But, the properties of neural networks were analyzed by machine learners, mathematicians and statisticians. After that, the algorithms were improved and the methodology stabilized.
- Neural networks resurfaced after 2010 with the new name deep learning.
- The major reason for the successes of deep learning is the availability of extremely large training datasets.
- It can be a solution to some niche problems such as image and video classification, speech and text modeling.

Introduction to Deep Learning

- Deep learning tools are able to imitate the functioning of the human brain for processing data and identify patterns for decision making.
- Deep learning algorithms help businesses to develop models that can predict more accurate outcomes to help them make better decisions.
- Most useful deep learning tools in 2024
  1. TensorFlow
  2. Pytorch
  3. Keras
  4. OpenNN
  5. Theano
  6. CNTK
  7. DeepLearningKit

  ⋮

Deep Learning in R

- First, install tensorflow R package from Github.

```
install.packages("remotes")
remotes::install_github("rstudio/tensorflow")
```

- Make sure that python is installed on your system.
- Next, install TensorFlow. This will automatically create an isolated virtual environment named "r-tensorflow".

```
library(tensorflow)
install_tensorflow(envname = "r-tensorflow")
```

- Alternatively, you can also install Keras, which installs TensorFlow.

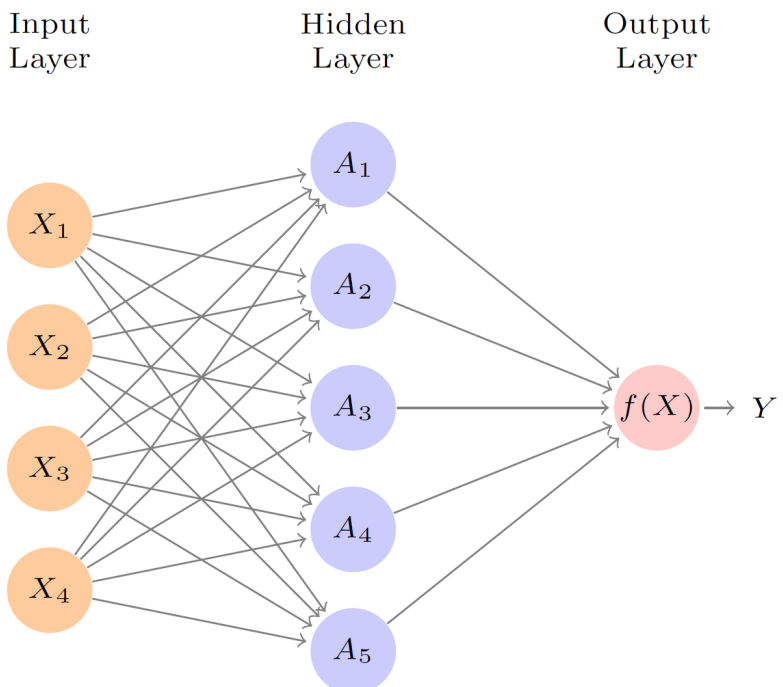
```
install.packages("keras")
library(keras)
install_keras()
```

Single Layer Neural Networks

- A neural network takes an input vector of p variables

$$X = (X_1, X_2, \ldots, X_p)$$

  and a nonlinear function $f(X)$ to predict the response Y.
- A neural network with a single hidden layer models a quantitative response using $p = 4$ predictors.



Single Layer Neural Networks

- The single layer neural networks consists of
  - p features $X_1, \ldots, X_p$ in the input layer
  - K hidden units in the hidden layer
  - $f(X)$ in the output layer
- The mathematical model of the neural network is

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

- The K activations A_k for $k = 1, \ldots, K$ in the hidden layer are

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j),$$

where $g(\cdot)$ is a nonlinear activation function that is specified in advance.

Single Layer Neural Networks

- Each A_k is a different transformation $h_k(X)$ of the original features, similar to the basis function in nonlinear models.
- Once A_k is computed, the neural network model is simply reduced to a multiple linear regression model.
- All the parameters $\beta_0, \ldots, \beta_K$ and $w_{10}, w_{11}, \ldots, w_{Kp}$ should be estimated from data.
- The total number of parameters is

$$(K+1) + K(p+1) = K(p+2) + 1.$$

- The name neural network originally derived from thinking of the hidden units as analogous to neurons in the brain.
  - A_k is close to 1 if firing
  - A_k is close to 0 if silent

Activation Functions

- The sigmoid activation function

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

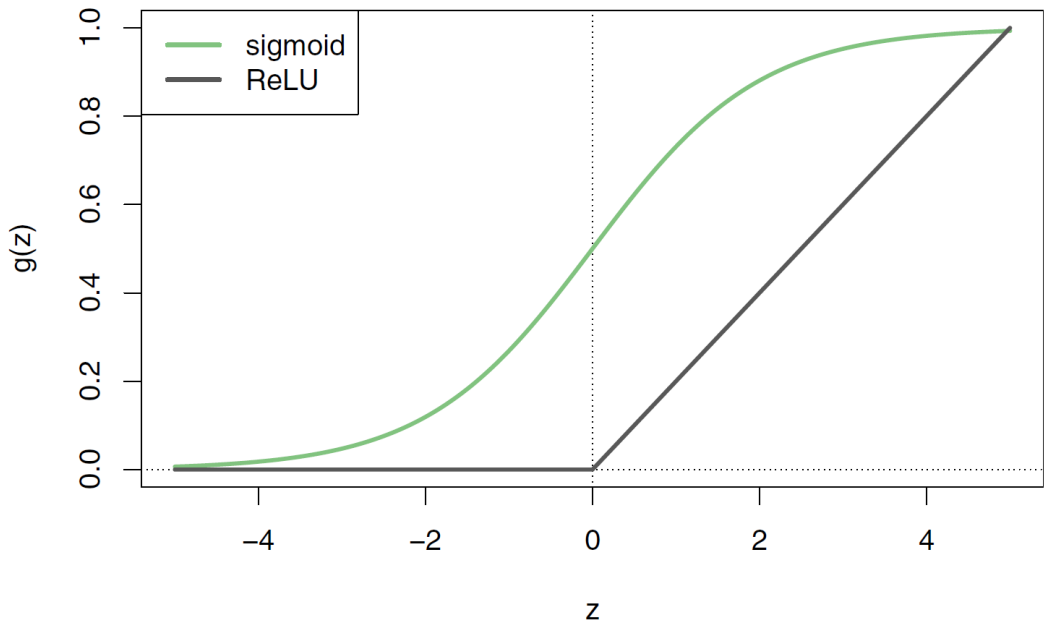
  which is equivalent to the inverse of logit function.
- The ReLU (rectified linear unit) activation function

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

- A ReLU activation can be computed and stored more efficiently than a sigmoid activation.

Activation Functions

- Sigmoid vs. ReLU



- The nonlinearity in the activation function $g(\cdot)$ is essential.
- In practice, we would not use a quadratic function for $g(z)$, since we would always get a second-degree polynomial.
- The sigmoid or ReLU activations do not have such a limitation.

Neural Network Parameters

- A neural network requires estimating the unknown parameters

$$\beta = (\beta_0, \beta_1, \ldots, \beta_K), \quad w_k = (w_{k0}, w_{k1}, \ldots, w_{kp})$$

  for $k = 1, 2, \ldots, K$.
- For a quantitative response, squared-error loss is used.
- The parameters are chosen to minimize

$$\sum_{i=1}^{n} (y_i - f(x_i))^2,$$

  where

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

Example

- A single layer neural network with $p = 2$ and $K = 3$

$$x_i = (2.0, -0.5)^{\mathrm{T}}, \ (\beta_0, \beta_1, \beta_2, \beta_3) = (1.0, -0.2, 0.5, 1.3)^{\mathrm{T}}$$

and

$$W = \begin{pmatrix} w_{10}, w_{11}, w_{12} \\ w_{20}, w_{21}, w_{22} \\ w_{30}, w_{31}, w_{32} \end{pmatrix} = \begin{pmatrix} 0.7, 1.2, -0.3 \\ 0.1, 2.1, -0.7 \\ -0.5, -1.5, 0.3 \end{pmatrix}$$

- The output $f(x_i)$ is then

$$f(x_i) = \beta_0 + \sum_{k=1}^{3} \beta_k \cdot g\left(w_{k0} + \sum_{j=1}^{2} w_{kj} x_{ij} \right).$$

  - $f(x_i) = 1.3357$ if $g(\cdot)$ is a sigmoid.
  - $f(x_i) = 2.6750$ if $g(\cdot)$ is a ReLU.

Example

```
x <- c(1, 2, -0.5)
beta <- c(1.0, -0.2, 0.5, 1.3)
W <- matrix(c(0.7, 1.2, -0.3, 0.1, 2.1, -0.7, -0.5, -1.5, 0.3),
            nrow=3, ncol=3, byrow=TRUE)
```

```
g1 <- function(x) c(1, 1/(1+exp(-x)))
g2 <- function(x) c(1, ifelse(x>=0, x , 0))
```

```
WX <- W %*% x
g1.WX <- g1(WX)
g2.WX <- g2(WX)
```

```
fx1 <- sum(beta * g1.WX)
fx2 <- sum(beta * g2.WX)
c(fx1, fx2)
```

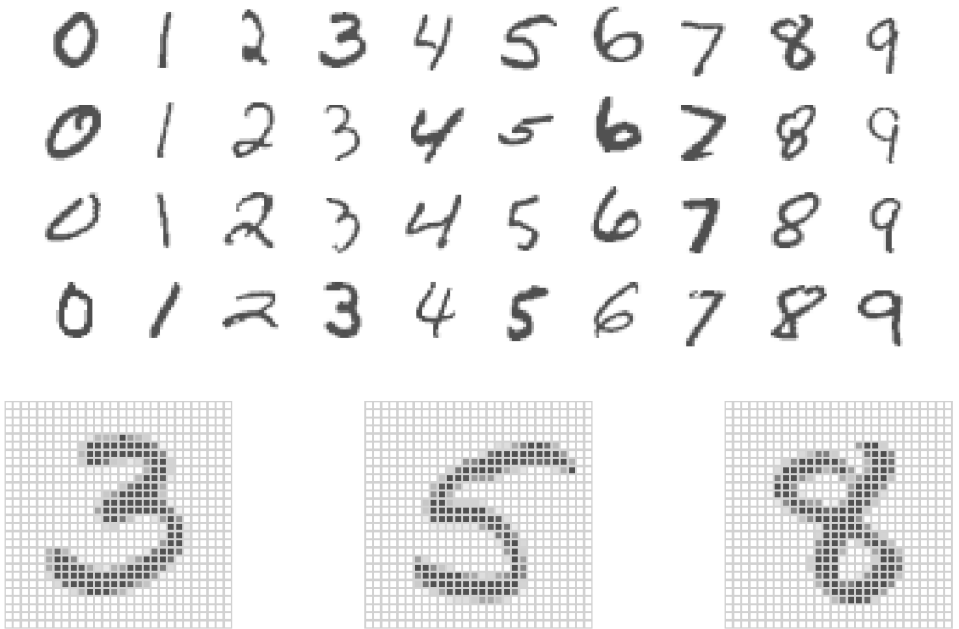
Multi-layer Neural Networks

- Modern neural networks typically have more than one hidden layer and many units per layer.
- In theory, a single hidden layer with a large number of units has the ability to approximate most functions.
- The learning task of discovering a good solution is made much easier with multiple layers each of modest size.
- Hand-written digit dataset: `MNIST`
  - Every image has $p = 28 \times 28 = 784$ pixels.
  - Each pixel is an eight-bit grayscale value between 0 and 255.
  - The class labels are 0 to 9 (10 dummy variables)

$$Y = (Y_0, Y_1, \ldots, Y_9)^{\mathrm{T}}, \qquad Y_j \in \{0, 1\}$$

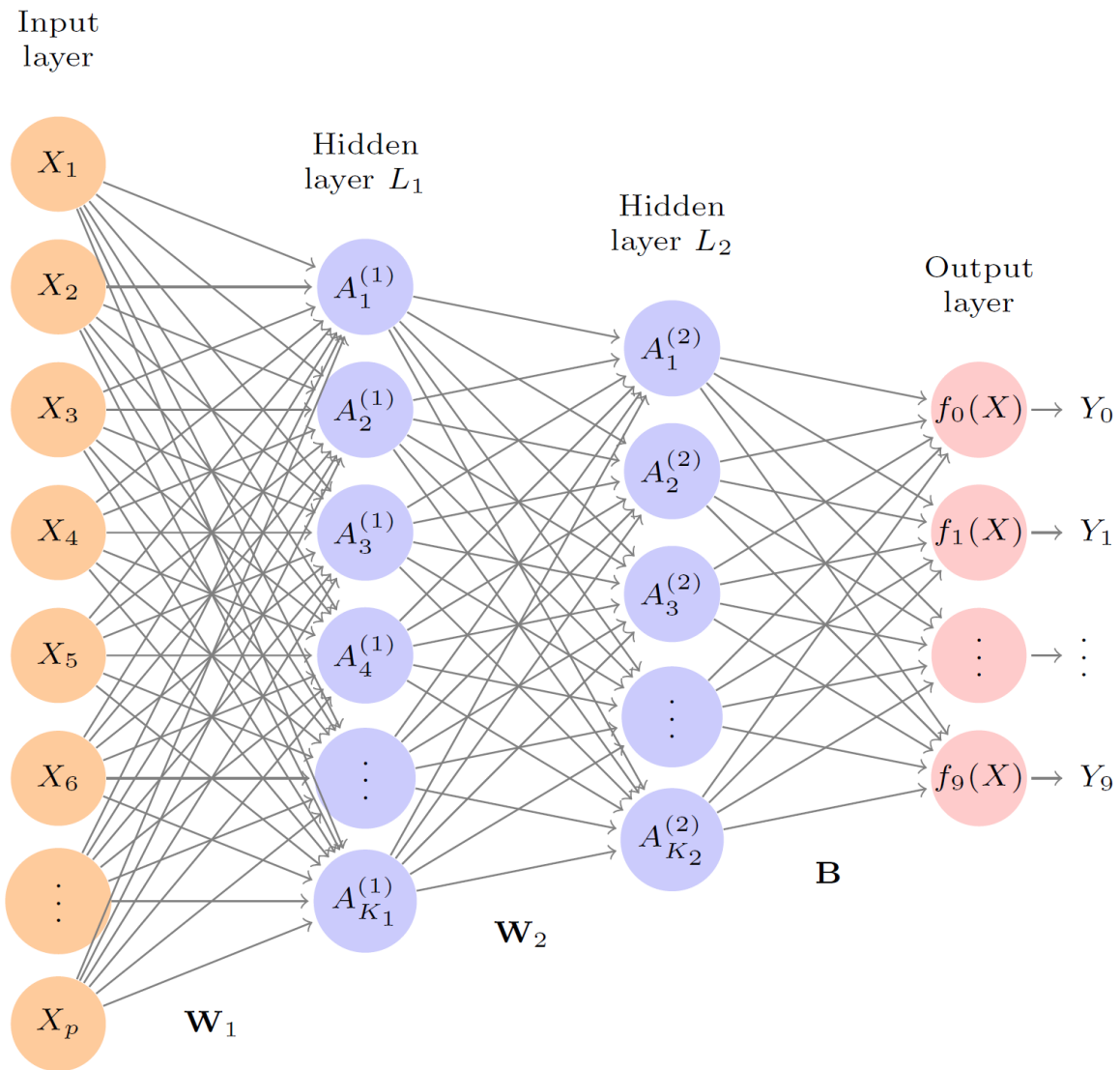
  - There are $60,000$ training images, and $10,000$ test images

Hand-written Digit Dataset



- Digit recognition problems were the catalyst that encoding accelerated the development of neural network technology in the late 1980s at AT&T Bell Laboratories and elsewhere.
- These tasks are not so simple for machines, and it has taken more than 30 years to refine the neural-network architectures to match human performance.

Multi-layer Neural Networks



Multi-layer Neural Networks

- It has two hidden layers L_1 and L_2, and the first layer is

$$A_k^{(1)} = h_k^{(1)}(X)$$
$$= g(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j)$$

for $k = 1, 2, \ldots, K_1$. The second layer is

$$A_l^{(2)} = h_l^{(2)}(X)$$
$$= g(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lk}^{(2)} A_k^{(1)})$$

for $l = 1, 2, \ldots, K_2$.

Multi-layer Neural Networks

- Through a chain of transformations, the network is able to build up fairly complex transformations of X.
- Just like in a multinomial logistic regression, we use softmax activation function

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}},$$

where

$$\begin{aligned} Z_m &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} h_l^{(2)}(X) \\ &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)}, \end{aligned}$$

for $m = 0, 1, \ldots, 9$.

Multi-layer Neural Networks

- Even though the goal is to build a classifier, our model actually estimates a probability for each of the 10 classes.
- Since the response is qualitative, we minimize the negative multinomial log-likelihood

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log\left(f_m(x_i)\right)$$

  to estimate coefficients.
- The objective is known as the cross-entropy.
- This is a generalization of the criterion for two-class logistic regression.
- If the response were quantitative, we would minimize the squared-error loss.

Multi-layer Neural Networks

- The number of parameters that need to be estimated are
  - $\boldsymbol{W}_1 : (p+1) \times K_1 = (784+1) \times 256 = 200,960$
  - $\boldsymbol{W}_2 : (K_1+1) \times K_2 = (256+1) \times 128 = 32,896$
  - $\boldsymbol{\beta} : (K_2+1) \times 10 = (128+1) \times 10 = 1,290.$
  - The total is

$$200,960 + 32,896 + 1,290 = 235,146.$$

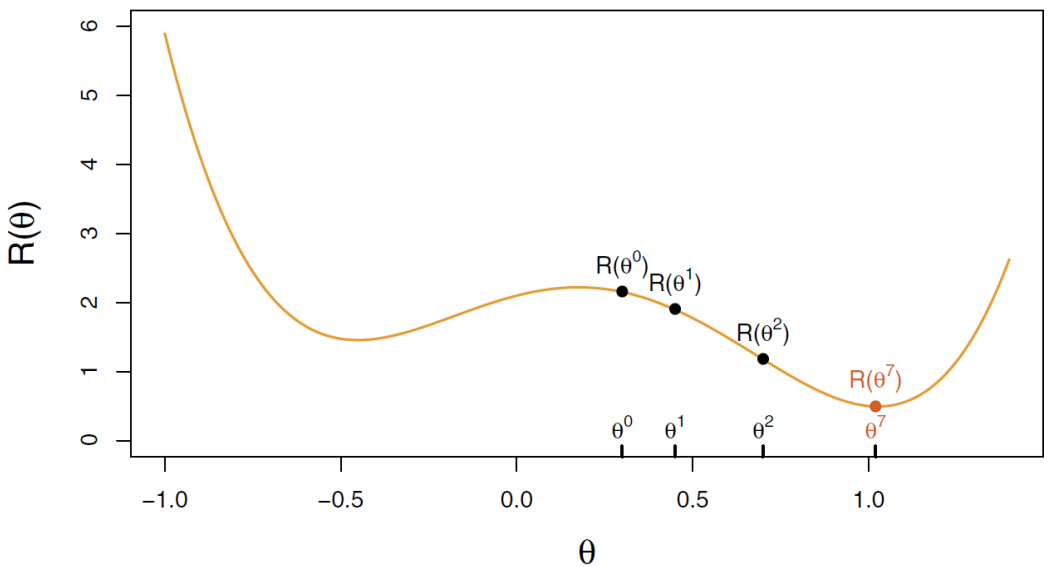
- A multinomial logistic regression requires

$$p \times (10-1) = 785 \times 9 = 7,065$$

  parameters.
- There are $60,000$ images in the training set.
  - Approximately $1/4$ of the number of parameters.
  - Some regularization is needed.

Estimation of Neural Network Parameters

- It is not straightforward to minimize, because of
  - The nested arrangement of the parameters
  - The symmetry of the hidden units
  - Nonconvex in the parameters
- There are two solutions:
  - A local minimum
  - A global minimum



Fitting Neural Network

- To overcome some of issues and to protect from overfitting, two general strategies are employed when fitting neural networks.
  - Slow learning
  - Regularization
- Slow learning:
  - The model is fit in a somewhat slow iterative fashion, using gradient descent.
  - The fitting process is then stopped when gradient overfitting is detected.
- Suppose we represent all the parameters in one long vector θ. Then, we want to minimize the objective

$$R(\theta) = \frac{1}{2} \sum_{i=1}^{n} \left(y_i - f_\theta(x_i) \right)^2 .$$

Gradient Descent

- The idea of gradient descent
  1. Start with a guess θ^0 for all parameters in θ, and set $t = 0$.
  2. Iterate until the objective fails to decrease
     (a) Find a vector δ that reflects a small change in θ, such that

$$\theta^{t+1} = \theta^t + \delta$$

reduce the objective; i.e.,

$$R(\theta^{t+1}) < R(\theta^t)$$

     (b) $t \longleftarrow t + 1$
- The goal is to get to the bottom through a series of steps.
- With our lucky starting guess θ^0, we can end up at the global minimum.
- In general, we can hope to end up at a good local minimum.

Gradient Descent

- The gradient of $R(\theta)$, evaluated at some current value gradient $\theta = \theta^m$, is the vector of partial derivatives at that point:

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

- The gradient gives the direction in θ-space in which $R(\theta)$ increases most rapidly.
- The idea of gradient descent is to move θ a little in the opposite direction:

$$\theta^{m+1} \longleftarrow \theta^m - \rho \nabla R(\theta^m),$$

  where ρ is a learning rate and typically small.
- If the gradient vector is zero, then we may have arrived at a minimum of the objective.

Chain Rule of Differentiation

- How complicated is the calculation of $\nabla R(\theta^m)$?
- It is quite simple due to chain rule of differentiation.
- We have

$$\begin{aligned} R(\theta) &= \sum_{i=1}^{n} R_i(\theta) \\ &= \frac{1}{2} \sum_{i=1}^{n} \left(y_i - f_\theta(x_i)\right)^2 . \end{aligned}$$

- For the i-th sample,

$$R_i(\theta) = \frac{1}{2} \left(y_i - \beta_0 - \sum_{k=1}^{K} \beta_k g(z_{ik}) \right)^2 ,$$

where $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$.

Chain Rule of Differentiation

- The partial derivative with respect to β_k

$$\frac{\partial R_i(\theta)}{\partial \beta_k} = \frac{\partial R_i(\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} = -(y_i - f_\theta(x_i)) \cdot g(z_{ik}).$$

- The partial derivative with respect to w_{kj}

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

- Notice that both partial derivatives contain the residual

$$r_i = y_i - f_\theta(x_i)$$

Backpropagation

- A fraction of that residual gets attributed to each of the hidden units according to the value of $g(z_{ik})$ in the partial derivative with respect to β_k.
- The act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule — a process known as backpropagation in the neural network literature.
- Although these calculations are straightforward, it takes careful bookkeeping to keep track of all the pieces.
- Gradient descent usually takes many steps to reach a local minimum.
- In practice, there are a number of approaches for accelerating the process.

Computational Methods of Gradient Descent

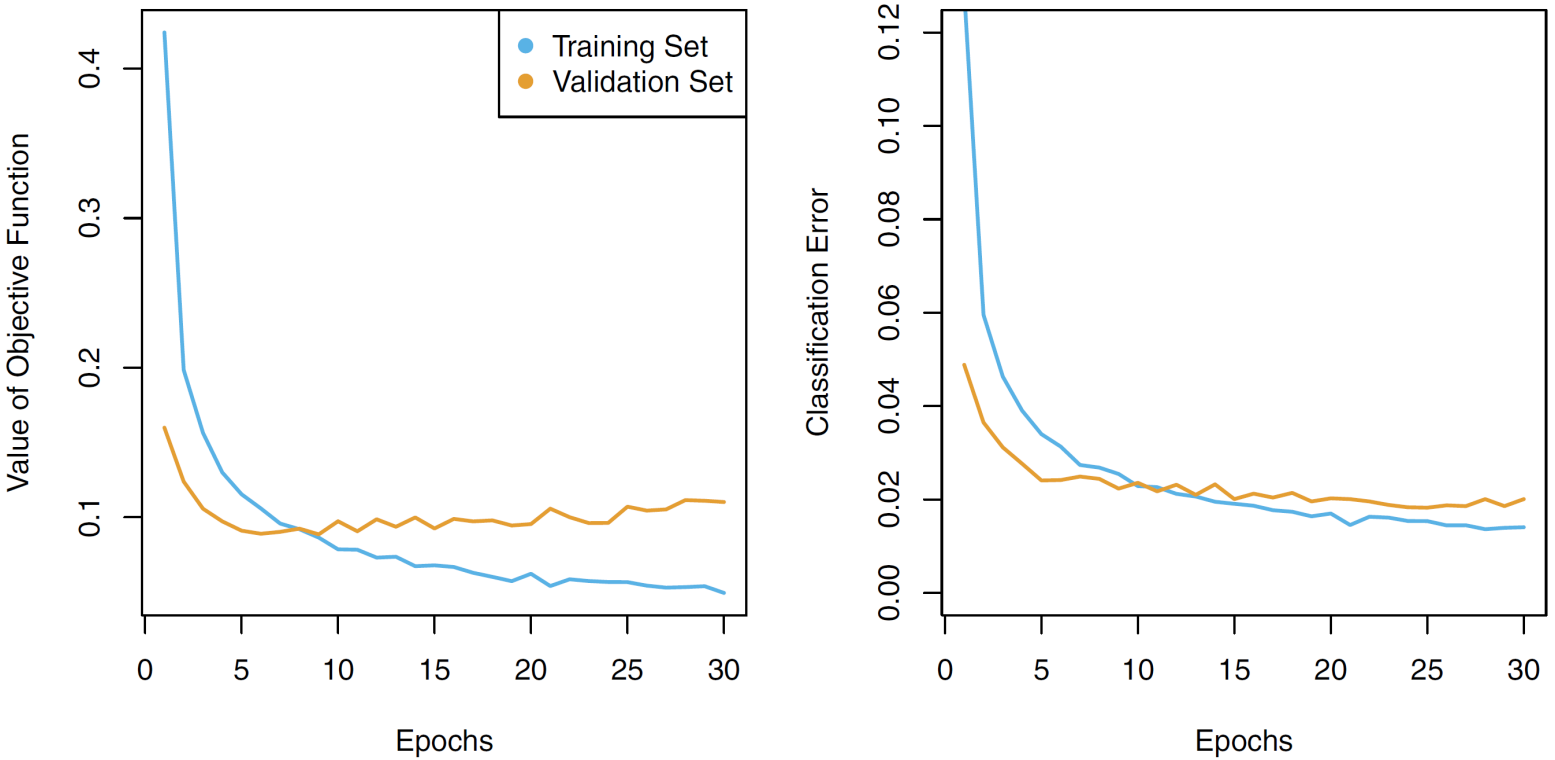
- Stochastic gradient descent (SGD).
  - We consider just one example at a time to take a single step.
  - It will never reach the minima but will keep dancing around it.
- Mini-batch gradient descent
  - We can sample a small fraction or mini-batch of them each time we compute a gradient step.
- Batch gradient descent
  - All the training data is taken into consideration to take a single step.
  - Computationally infeasible when n is very large.

Computational Methods of Gradient Descent

- One epoch typically means the algorithm sees every training data once.
  - In batch gradient descent, every epoch the parameters are updated once.
  - In mini-batch gradient descent, every epoch the parameters are updated about n/b times if the mini-batch size is b.
  - In SGD, every epoch the parameters are updated about n times.
- In `MNIST` dataset,
  - The mini-batch size was 128 per gradient update.
  - 20% of training set ($n = 60,000$) was used for validation set.
  - Only $48,000$ observations was used for training.
  - Approximately $48,000/128 \approx 375$ gradient updates per epoch.

Computational Methods of Gradient Descent

- The term epochs counts the number of times an equivalent of epochs the full training set has been processed.
- The validation objective starts to increase by 30 epochs.
- Early stopping can be used as a form of regularization.



Regularization

- When the number of parameters is much greater than the sample size, regularization is essential to avoid overfitting.
- With a ridge penalty,

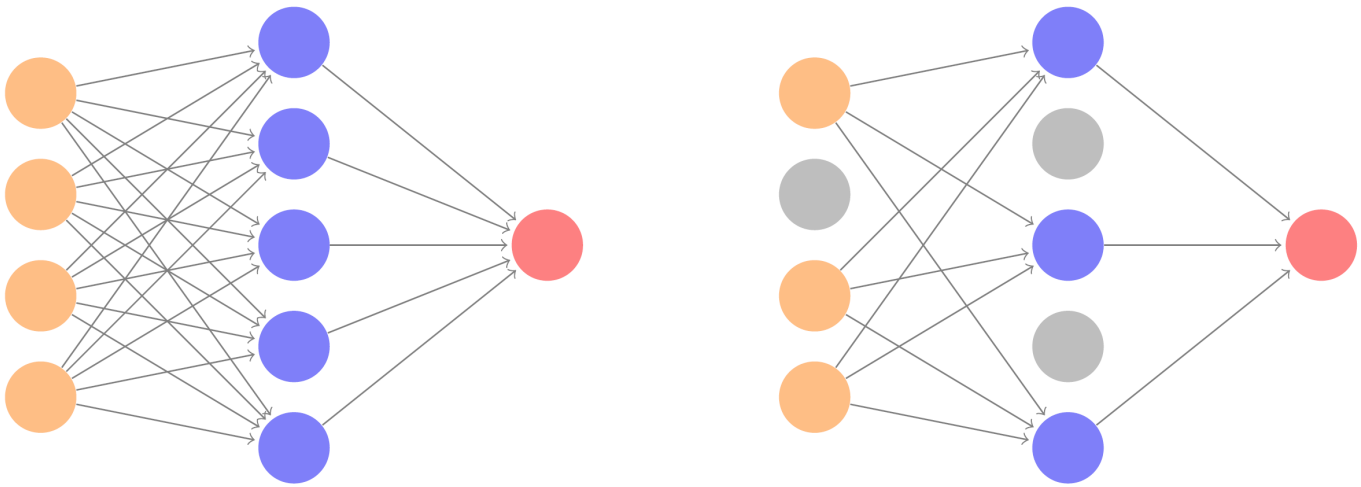
$$R(\theta;\lambda) = -\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im}\log\left(f_m(x_i)\right) + \lambda\sum_{j}\theta_j^2,$$

  where λ is a pre-defined value.
- We can also use different values of λ for the groups of weights from different layers. For example,
  - Penalized $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ if λ is large.
  - Non-penalized $\boldsymbol{\beta}$ if λ is small.
- Lasso regularization is also popular as an alternative to ridge.

Dropout Regularization

- Dropout regularization is to randomly remove a fraction ϕ of the units in a layer when fitting the model, inspired by random forest.
- This is done separately each time a training observation is processed.
- The weights of the surviving units are scaled up by a factor of $1/(1-\phi)$ to compensate.
- It prevents nodes from becoming over-specialized, and can be seen as a form of regularization.
- In practice, dropout is achieved by randomly setting the activations for the "dropped out" units to zero.

Dropout Regularization



- Test error rate on the MNIST data

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

Neural Network Tuning

- The number of hidden layers and units per layer
  - The number of units per hidden layer can be large.
  - Overfitting can be controlled via the various forms of regularization
- Regularization tuning parameters
  - The dropout rate ϕ and λ for ridge and lasso.
  - They are typically set separately at each layer.
- Details of gradient descent
  - The batch size, the number of epochs, etc
- Choices such as these can make a huge difference.
- In `MNIST` data, misclassification error rate was 1.8% $\sim$ 2.3%.
- The tinkering process can be tedious and result in overfitting if done carelessly.

Example: Hitters Data

```
library(ISLR)
data(Hitters)
str(Hitters)
```

```
summary(Hitters)
sum(is.na(Hitters))
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
```

```
set.seed(1234)
test <- sample(1:n, floor(n/3))
```

```
g <- lm(Salary ~ ., data=Gitters[-test, ])
pred <- predict(g, Gitters[test, ])
```

```
with(Gitters[test, ], mean(abs(pred - Salary)))
r0 <- mean(abs(pred - Gitters[test, "Salary"]))
```

```
library(glmnet)
x0 <- model.matrix(Salary ~ ., data=Gitters)
x <- scale(x0[,-1])
dim(x)
```

```
y <- Gitters$Salary
length(y)
```

```
set.seed(1234)
gcv <- cv.glmnet(x[-test , ], y[-test], type="mae")
```

```
coef.min <- as.numeric(coef(gcv, s="lambda.min"))
coef.1se <- as.numeric(coef(gcv, s="lambda.1se"))
data.frame(OLS=g$coef, coef.min, coef.1se)
```

```
pred1 <- predict(gcv, x[test, ], s="lambda.min")
pred2 <- predict(gcv, x[test, ], s="lambda.1se")
```

```
r1 <- mean(abs(y[test] - pred1))
r2 <- mean(abs(y[test] - pred2))
```

```
w1 <- which(coef.min[-1]!=0)
w2 <- which(coef.1se[-1]!=0)
ww <- which(colnames(Gitters)=="Salary")
```

```
Gitters1 <- Gitters[,c(ww,w1)]
Gitters2 <- Gitters[,c(ww,w2)]
```

```
g3 <- lm(Salary ~ ., data=Gitters1[-test, ])
g4 <- lm(Salary ~ ., data=Gitters2[-test, ])
pred3 <- predict(g3, Gitters1[test, ])
pred4 <- predict(g4, Gitters2[test, ])
```

```
r3 <- mean(abs(pred3 - Gitters1[test, "Salary"]))
r4 <- mean(abs(pred4 - Gitters2[test, "Salary"]))
```

```
set.seed(1234)
gcv0 <- cv.glmnet(x[-test , ], y[-test], alpha=0, type="mae")
pred5 <- predict(gcv0, x[test, ], s="lambda.min")
pred6 <- predict(gcv0, x[test, ], s="lambda.1se")
```

```
r5 <- mean(abs(y[test] - pred5))
r6 <- mean(abs(y[test] - pred6))
```

```
out <- matrix(c(r0, r1, r2, r3, r4, r5, r6), ncol=1)
rownames(out) <- c("lm", "lasso.min", "lasso.1se",
                   "lasso.min+lm", "lasso.1se+lm",
                   "ridge.min", "ridge.1se")
colnames(out) <- "MAE"
out
```

```
data.frame(out, ranking=rank(out))
```

Fitting the Neural Network

- Our neural network model has a single hidden layer with 50 hidden units and a ReLU activation function with the dropout rate of 40%.
- The output layer has just one unit with no activation function, indicating that the model provides a single quantitative output.

```
library(tensorflow)
library(keras)
library(ggplot2)
```

```
modnn <- keras_model_sequential() %>%
         layer_dense(units=50, activation="relu",
                     input_shape=ncol(x)) %>%
         layer_dropout(rate=0.4) %>%
         layer_dense(units=1)
```

Fitting the Neural Network

- Next, we add details to `modnn` that control the fitting algorithm where we minimize squared-error loss.

```
compile(modnn, loss="mse", optimizer="rmsprop",
        metrics=list("mean_absolute_error"))
```

- We supply the training and validation data and two fitting parameters, `epochs` and `batch_size`.

```
history <- fit(modnn, x[-test, ], y[-test],
               epochs=1000, batch_size=32,
               validation_data=list(x[test , ], y[test ]))
```

- The algorithm randomly selects 32 training observations for the computation of the gradient.
- Since the training set has $n = 176$, an epoch is $176/32 = 5.5$ SGD steps

Fitting the Neural Network

- We can plot the `history` to display the mean absolute error for the training and test data.

```
plot(history)
```

- If run the `fit()` command a second time in the same R session, then the fitting process will pick up where it left off.
- Due to the use of SGD, the prediction results vary slightly with each fit.
- Unfortunately, the `set.seed()` function does not ensure identical results.

```
prednn <- predict(modnn, x[test, ])
r7 <- mean(abs(y[test] - prednn))
out <- rbind(out, r7)
rownames(out)[8] <- "neural.net"
data.frame(out, ranking=rank(out))
```

Example: MNIST Data

- Remind of Hand-written digit dataset: `MNIST`
  - Every image has $p = 28 \times 28 = 784$ pixels.
  - Each pixel is an eight-bit grayscale value between 0 and 255.
  - The class labels are 0 to 9 (10 dummy variables)

$$Y = (Y_0, Y_1, \ldots, Y_9)^{\mathrm{T}}, \qquad Y_j \in \{0, 1\}$$

  - There are $60,000$ training images, and $10,000$ test images
- The `keras` package comes with a number of example datasets, including the `MNIST` digit data.
- The images are stored as a three-dimensional array, so we need to reshape them into a matrix.
- Also, we need to "one-hot" encode the class label.
- Neural networks are somewhat sensitive to the scale of the inputs, so we rescale eight-bit grayscale values to the unit interval.

```
mnist <- dataset_mnist()
```

```
x_train <- mnist$train$x
g_train <- mnist$train$y
x_test <- mnist$test$x
g_test <- mnist$test$y
rbind(dim(x_train), dim(x_test))
rbind(dim(g_train), dim(g_test))
```

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
rbind(dim(x_train), dim(x_test))
```

```
summary(as.numeric(x_train))
summary(as.numeric(x_test))
x_train <- x_train/255
x_test <- x_test/255
```

```
rbind(dim(g_train), dim(g_test))
range(as.numeric(g_train))
range(as.numeric(g_test))
```

```
y_train <- to_categorical(g_train, 10)
y_test <- to_categorical(g_test, 10)
rbind(dim(y_train), dim(y_test))
range(as.numeric(y_train))
range(as.numeric(y_test))
```

```
modelnn <- keras_model_sequential() %>%
           layer_dense(units=256, activation="relu",
                       input_shape=784) %>%
           layer_dropout(rate=0.4) %>%
           layer_dense(units=128, activation="relu") %>%
           layer_dropout(rate=0.3) %>%
           layer_dense(units=10, activation="softmax")
summary(modelnn)
```

```
compile(modelnn, loss="categorical_crossentropy",
        optimizer="rmsprop", metrics=c("accuracy"))
```

```
system.time(
  history <- fit(modelnn, x_train, y_train, epochs=30,
                 batch_size=128, validation_split=0.2)
)
plot(history)
```

```
accuracy <- function(pred, truth) {
              mean(drop(as.numeric(pred))==drop(truth))
            }
```

```
modelnn %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
pred1 <- predict(modelnn, x_test)
accuracy(k_argmax(pred1), g_test)
```

```
modelnn2 <- keras_model_sequential() %>%
            layer_dense(input_shape=784, units=10,
                        activation="softmax")
summary(modelnn2)
```

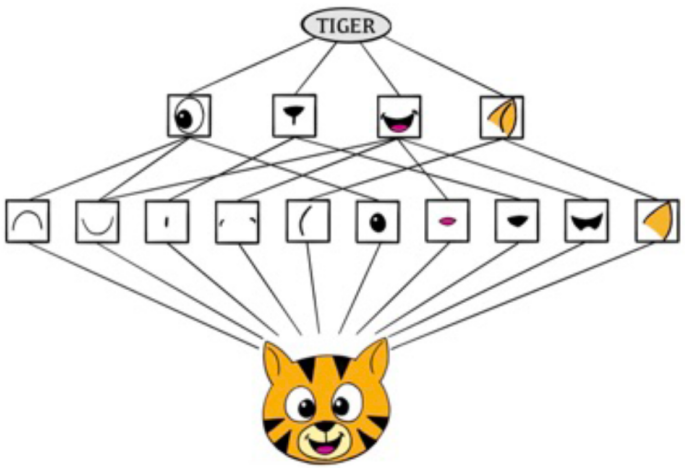
```
compile(modelnn2, loss="categorical_crossentropy",
        optimizer="rmsprop", metrics = c("accuracy"))
```

```
history2 <- fit(modelnn2, x_train, y_train, epochs=30,
                batch_size=128, validation_split=0.2)
plot(history2)
```

```
pred2 <- predict(modelnn2, x_test)
accuracy(k_argmax(pred2), g_test)
```

Convolutional Neural Networks

- A special family of convolutional neural networks (CNNs) has evolved for classifying images and has shown spectacular success on a wide range of problem.
- CNNs mimic how humans classify images, recognizing specific features or patterns anywhere in the image.
- CNNs first identifies low-level features in the input image and then they are combined to form higher-level features.
- The presence or absence of the higher-level features contributes to the probability of any given output class.



Convolutional Neural Networks

- How does a convolutional neural network build up this hierarchy?
- CNNs combines two specialized types of hidden layers:
  - Convolution layers
  - Pooling layers
- Convolution layers search for instances of small patterns in the image, whereas pooling layers downsample these to select a prominent subset.
- In order to achieve state-of-the-art results, contemporary neural network architectures make use of many convolution and pooling layers.

Convolution Layers

- A convolution layer is made up of a large number of convolution filters.
- A convolution filter basically amounts to repeatedly multiplying matrix elements and then adding the results.

Original Image $\times$ Convolution Filter $=$ Convolved Image

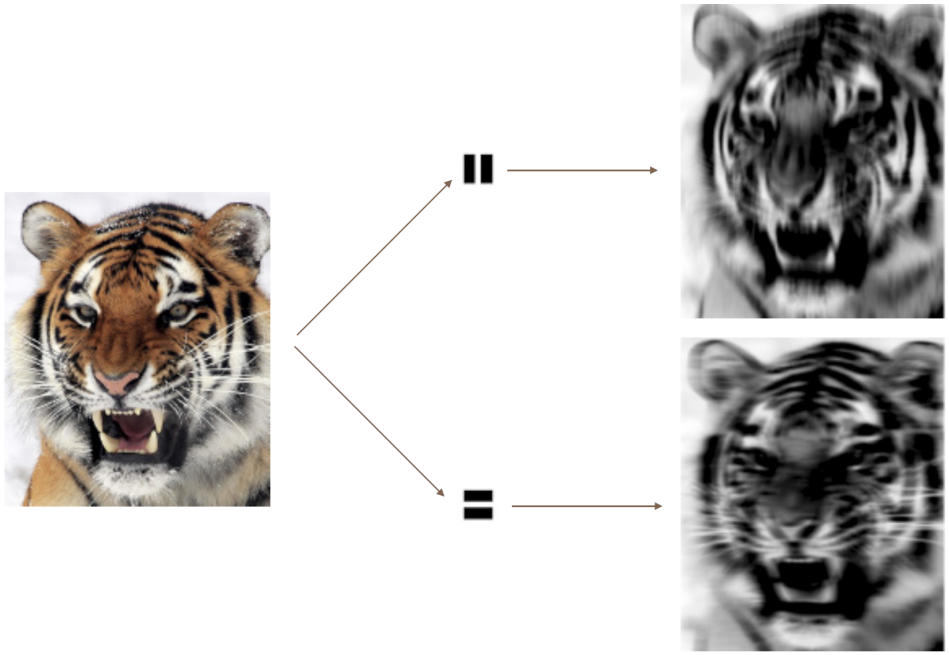
$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{pmatrix} \times \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} =$$

$$\begin{pmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{pmatrix}$$

Convolution Layers

- The convolution filter is applied to every 2×2 submatrix of the original image in order to obtain the convolved image.
- If a 2×2 submatrix of the original image resembles the convolution filter, then it will have a large value in the convolved image; otherwise, it will have a small value.
- The convolved image highlights regions of the original image that resemble the convolution filter.
- In general, convolution filters are small $l_1 \times l_2$ arrays, with l_1 and l_2 small positive integers that are not necessarily equal.
- In next figure, with a 192×179 image of a tiger each convolution filter is a 15×15 image containing mostly zeros (black), with a narrow strip of ones (white) oriented either vertically or horizontally within the image.

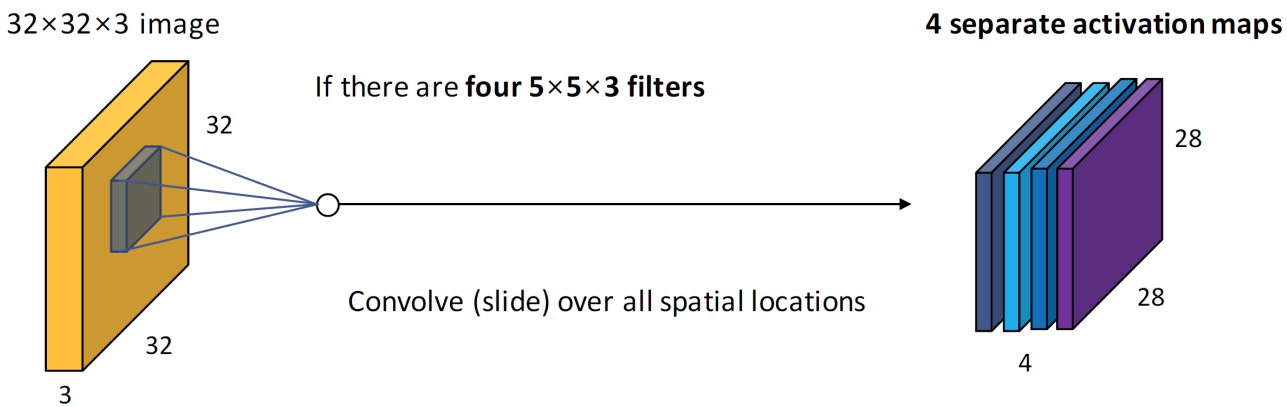
Convolution Layers



- The vertical stripe filter picks out vertical stripes and edges in the original image
- The horizontal stripe filter picks out horizontal stripes and edges in the original image.
- We use a whole bank of filters to pick out a variety of differently-oriented edges and shapes in the image.

Convolution Layers

- The filter weights are the parameters going from an input layer to a hidden layer, with one hidden unit for each pixel in the convolved image.
- The parameters are highly structured since there are many structural zeros.
- The same weights in a given filter are reused for all possible patches in the image (weight sharing).
- If we use K different convolution filters at this first hidden layer, we get K two-dimensional output feature maps, which together are treated as a single three-dimensional feature map.



Pooling Layers

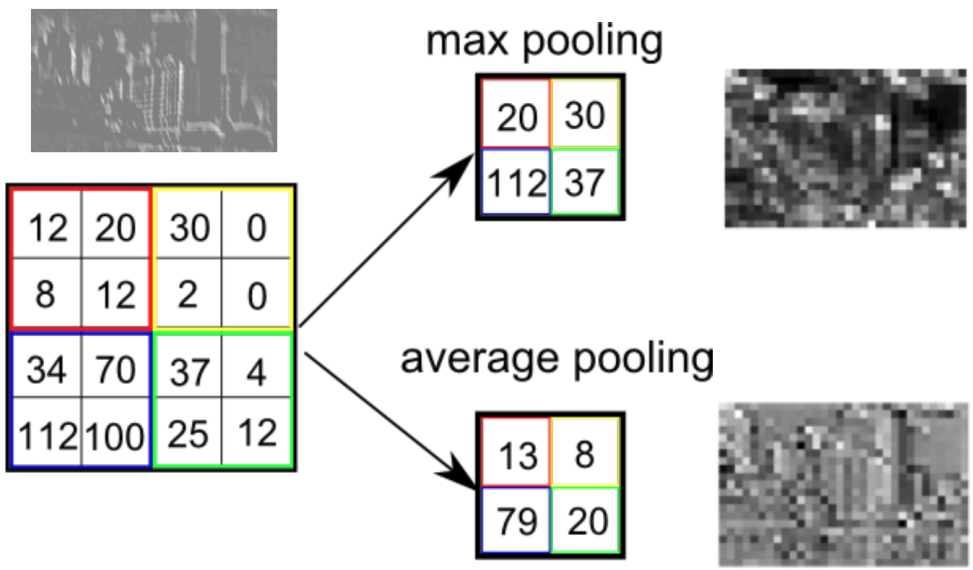
- A pooling layer provides a way to condense a large image into a smaller pooling.
- The max pooling summarizes each non-overlapping block of pixels in an image using the maximum value in the block.

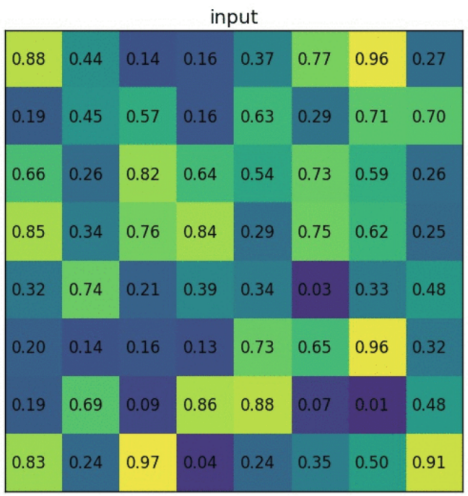
$$\begin{pmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{pmatrix} \longrightarrow \begin{pmatrix} 3 & 5 \\ 2 & 4 \end{pmatrix}$$

- Pooling layer essentially reduces the size of the image.
- It also provides some location invariance
  - As long as there is a large value in the block, the whole block registers as a large value in the reduced image.

Pooling Layers

- Types of pooling
  - Max pooling
  - Average pooling
  - Region of interest (ROI) pooling
  - Stochastic pooling

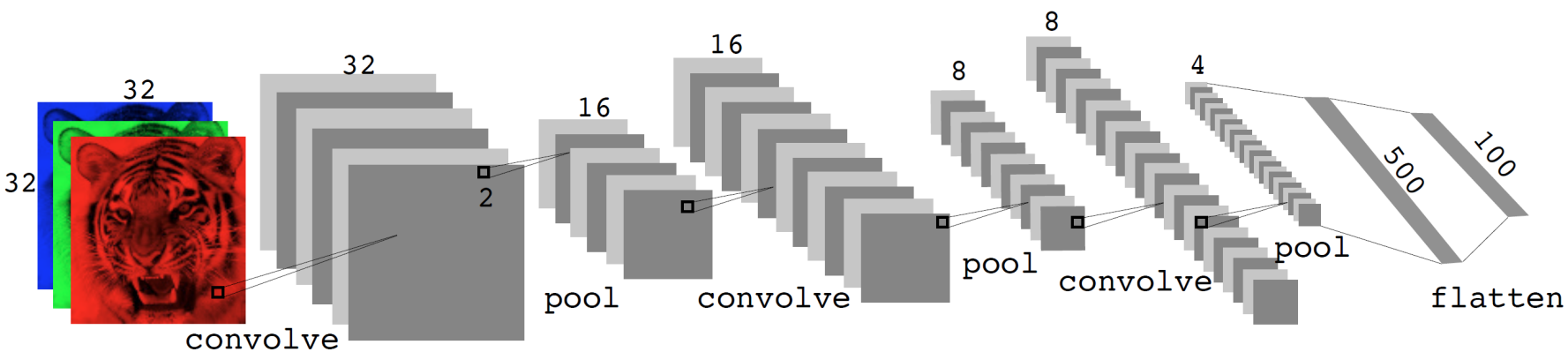




ROI pooling

Architecture of CNN

- The number of convolution filters is similar to the number of units at a particular hidden layer.
- Deep CNNs have many of both convolution and pooling layers, where convolve-then-pool sequence is repeated.
- The sequence is repeated until the pooling has reduced each channel feature map down to just a few pixels in each dimension, where the pixels are treated as separate unit (flattened).



Architecture of CNN

- Each subsequent convolve layer is similar to the first. It takes as input the three-dimensional feature map from the previous layer and treats it like a single multi-channel image.
- Since the channel feature maps are reduced in size after each pool layer, we usually increase the number of filters in the next convolve layer to compensate.
- Sometimes we repeat several convolve layers before a pool layer. This effectively increases the dimension of the filter.
- There are many tuning parameters to be selected in constructing a CNN, including the number, nature, and sizes of each layer.
- Dropout learning can be used at each layer, as well as lasso or ridge regularization.

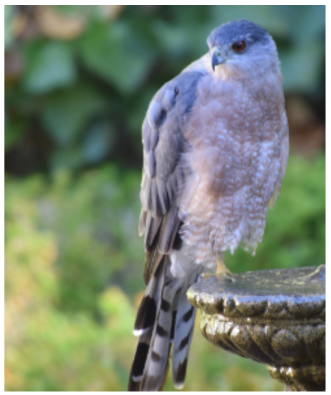
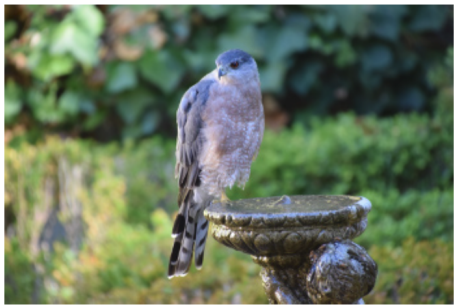
Data Augmentation

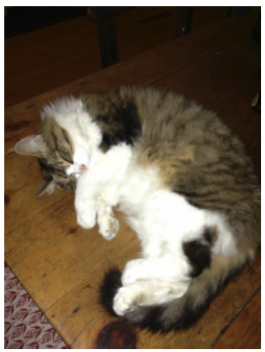
- An additional important trick used with image modeling is data augmentation.
- Each training image is replicated many times with each replicate randomly distorted in a natural way.
  - zoom, horizontal and vertical shift, shear, small rotations
- This is a way of increasing the training set to protect against overfitting.
- This is a form of regularization – we build a cloud of images around each original image, all with the same label.

Pretrained Classifier

- An industry-level pretrained classifier `resnet50` predicted the class of some new images. It was trained using the `imagenet` data set, which consists of millions of images.
- Much of the work in fitting a CNN is in learning the convolution filters at the hidden layer – these are the coefficients of a CNN.
- One can use the pretrained hidden layers for new problems with much smaller training sets (weight freezing) – just train the last few layers of the network, requiring much less data.
- Next example demonstrates the performance of `resnet50` on six photographs.
  - The true (intended) label at the top of each panel
  - The top three choices of the classifier (out of 100)
  - The numbers are the estimated probabilities for each choice

Pretrained Classifier




flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07
Lhasa Apso		**cat**		**Cape weaver**	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

Example: CIFAR100 Data

```
library(tensorflow)
library(keras)
```

```
cifar100 <- dataset_cifar100()
names(cifar100)
x_train <- cifar100$train$x; x_test <- cifar100$test$x
g_train <- cifar100$train$y; g_test <- cifar100$test$y
rbind(dim(x_train), dim(x_test))
rbind(dim(g_train), dim(g_test))
```

```
range(x_train[1,,,1])
x_train <- x_train/255
x_test <- x_test/255
```

```
range(g_train)
y_train <- to_categorical(g_train, 100)
dim(y_train)
range(y_train)
```

```
library(jpeg)
```

```
k <- 10
plot(x_train[k,,,])
plot(as.raster(x_train[k,,,]))
```

```
g_train[k,]
w <- which(g_train[,1]==g_train[k,])
g_train[w,]
```

```
w
par(mar = c(0, 0, 0, 0), mfrow = c(5, 5))
ww <- sample(w, 25)
for (i in 1:25) {
    plot(as.raster(x_train[ww[i],,, ]))
}
```

```
set.seed(123)
s <- sample(0:99, 1)
w <- which(g_train[,1]==s)
ww <- sample(w, 25)
for (i in 1:25) {
    plot(as.raster(x_train[ww[i],,, ]))
}
```

```
s <- sample(0:99, 1)
w <- which(g_train[,1]==s)
ww <- sample(w, 25)
for (i in 1:25) {
    plot(as.raster(x_train[ww[i],,, ]))
}
```

```
index <- sample(seq(50000), 25)
for (i in index) {
    plot(as.raster(x_train[i,,, ]))
}
```

```
model <- keras_model_sequential() %>%
        layer_conv_2d(filters=32, kernel_size=c(3, 3),
                      padding="same", activation="relu",
                      input_shape=c(32, 32, 3)) %>%
        layer_max_pooling_2d(pool_size=c(2, 2)) %>%
        layer_conv_2d(filters=64, kernel_size=c(3, 3),
                      padding="same", activation="relu") %>%
        layer_max_pooling_2d(pool_size=c(2, 2)) %>%
        layer_conv_2d(filters=128, kernel_size=c(3, 3),
                      padding="same", activation="relu") %>%
        layer_max_pooling_2d(pool_size=c(2, 2)) %>%
        layer_conv_2d(filters=256, kernel_size=c(3, 3),
                      padding="same", activation="relu") %>%
        layer_max_pooling_2d(pool_size=c(2, 2)) %>%
        layer_flatten() %>%
        layer_dropout(rate=0.5) %>%
        layer_dense(units=512, activation="relu") %>%
        layer_dense(units=100, activation="softmax")
```

```
summary(model)
```

```
compile(model, loss="categorical_crossentropy",
        optimizer="rmsprop", metrics = c("accuracy"))
```

```
#### It takes a couple of minutes ######################
history <- fit(model, x_train, y_train, epochs=30,
               batch_size=128, validation_split=0.2)
```

```
library(ggplot2)
plot(history)
```

```
pred <- predict(model, x_test)
accuracy(k_argmax(pred), g_test)
```

Example: Book Image Data

- Next, we show how to use a CNN pretrained on the `imagenet` database to classify natural image.
- We copied six jpeg images from a digital photo album into the directory `book_images`.
- We first read in the images, and convert them into the array format.
- We then load the trained network where the model has 50 layers, with a fair bit of complexity.
- Finally, we classify our six images, and return the top three class choices in terms of predicted probability for each.

```
url <- "https://www.statlearning.com/s/book_images.zip"
data <- download.file(url, "book_images.zip")
unzip("book_images.zip", exdir=".")
```

```
img_dir <- "book_images"
image_names <- list.files(img_dir)
num_images <- length(image_names)
x <- array(dim=c(num_images, 224, 224, 3))
```

```
for (i in 1:num_images) {
  img_path <- paste(img_dir, image_names[i], sep="/")
  img <- image_load(img_path, target_size=c(224, 224))
  x[i,,, ] <- image_to_array(img)
}
x <- imagenet_preprocess_input(x)
dim(x)
```

```
model <- application_resnet50(weights="imagenet")
summary(model)
```

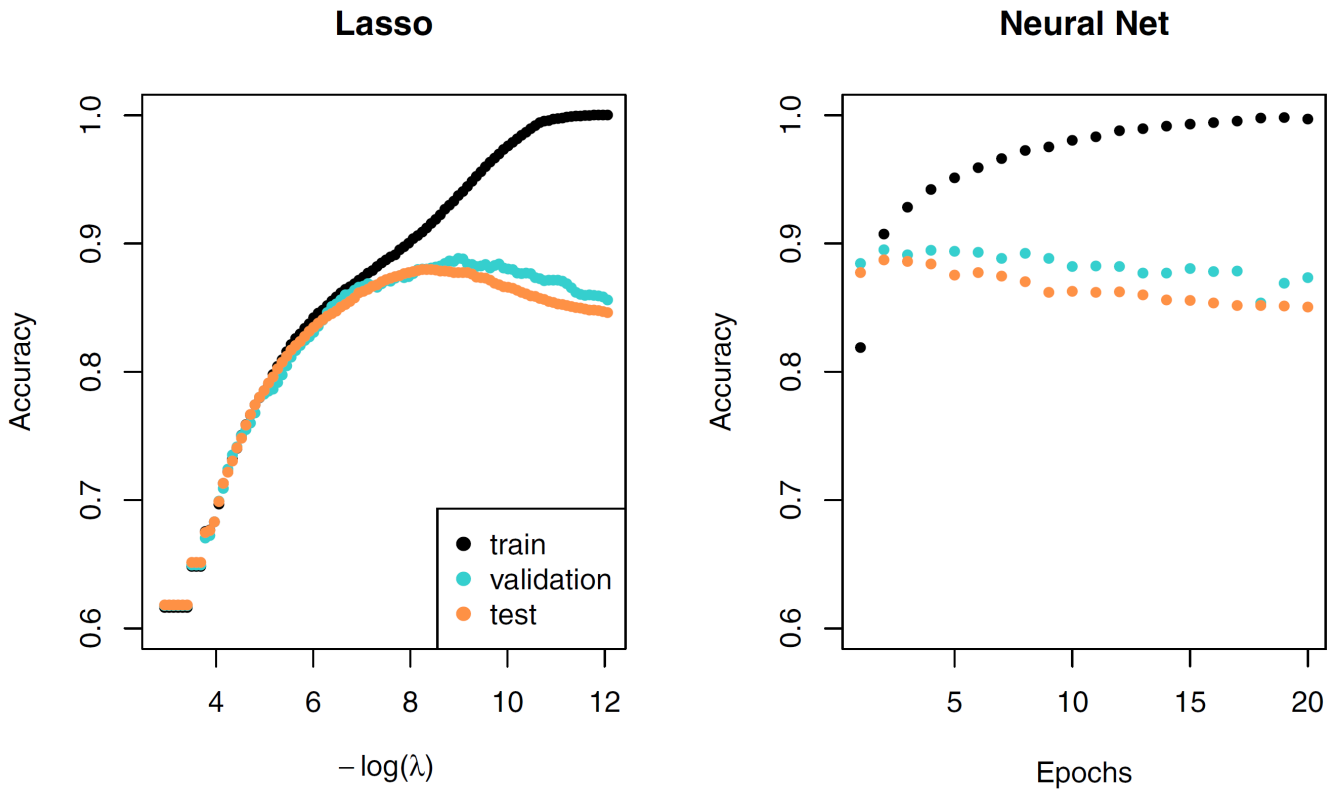
```
pred <- predict(model, x)
res <- imagenet_decode_predictions(pred, top=3)
names(res) <- image_names
print(res)
```

Document Classification

- Predicting attributes of documents has important applications in industry and science.
- Example of `IMDb` (Internet Movie Database) ratings
  - The response is either positive or negative of the review.
  - 25,000 documents for training set (2,000 validation set)
  - 25,000 documents for test set
- Need to find a way to featurize a document.
- The bag-of-words model score each document for the presence or absence of each of the words in a list.
  - The binary feature vector has length 10,000 with mostly of 0.
  - Training feature matrix X has $25,000 \times 10,000$, where only 1.3% nonzero.
  - The matrix can be stored efficiently in sparse matrix format.
- There are a variety of ways to account for the document length. For example, one could instead record the relative frequency of words.

Document Classification

- Two statistical models.
  - A lasso logistic regression – λ from validation set
  - A two-class neural network with two hidden layers, each with 16 ReLU units – training epochs from validation set



- Both models achieve a test-set accuracy of about 88%.

Document Classification

- The bag-of-words model summarizes a document by the words present, and ignores their context.
- There are at least two popular ways to take the context into account.
  - The bag-of-n-grams model: For example, a bag of 2-grams records the consecutive co-occurrence of every distinct pair of words. "Blissfully long" can be seen as a positive phrase in a movie review, while "blissfully short" a negative.
  - Treat the document as a sequence, taking account of all the words in the context of those that preceded and those that follow.
- For sequences of data, we need to model recurrent neural networks, which can be applied to weather forecasting, speech recognition, language translation, and time-series prediction.

Example: IMDb Data

- Now we perform document classification on the IMDb dataset.
- We limit the dictionary size to the 10,000 most frequently-used words and tokens.

```
max_features <- 10000
imdb <- dataset_imdb(num_words=max_features)
```

- Next command is a shortcut for unpacking the list of lists.

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb
c(length(x_train), length(y_train))
c(length(x_test), length(y_test))
```

- Each element of x_train is a vector of numbers between 0 and 9999, referring to the words found in the dictionary.

```
x_train[1:3]
x_train[[1]]
```

```
tran.values <- unlist(x_train)
test.values <- unlist(x_test)
summary(tran.values)
summary(test.values)
par(mfrow=c(1,2))
hist(tran.values, nclass=50, col="orange", main="Training set")
hist(test.values, nclass=50, col="purple", main="Test set")
```

```
tran.len <- sapply(x_train, length)
test.len <- sapply(x_train, length)
summary(tran.len)
summary(test.len)
hist(tran.len, nclass=50, col="orange", main="Training set")
hist(test.len, nclass=50, col="purple", main="Test set")
```

```
c(length(y_train), length(y_test))
table(y_train)
table(y_test)
```

```
word_index <- dataset_imdb_word_index()
length(word_index)
```

```
word <- names(word_index)
idx <- unlist(word_index, use.names=FALSE)
word[match(1:50, idx)]
```

```
decode_review <- function(text, word_index) {
    word <- names(word_index)
    idx <- unlist(word_index, use.names=FALSE)
    word <- c("<PAD>", "<START>", "<UNK>", "<UNUSED>", word)
    idx <- c(0:3, idx + 3)
    words <- word[match(text, idx, 2)]
    paste(words, collapse = " ")
}
```

```
decode_review(x_train[[1]], word_index)
decode_review(x_train[[2]], word_index)
decode_review(x_train[[25000]], word_index)
```

```
library(Matrix)
one_hot <- function(sequences, dimension) {
    seqlen <- sapply(sequences, length)
    n <- length(seqlen)
    rowind <- rep(1:n, seqlen)
    colind <- unlist(sequences)
    sparseMatrix(i=rowind, j=colind, dims=c(n, dimension))
}
```

```
x_train_1h <- one_hot(x_train, 10000)
x_test_1h <- one_hot(x_test, 10000)
```

```
dim(x_train_1h)
nnzero(x_train_1h) / (25000 * 10000)
nnzero(x_test_1h) / (25000 * 10000)
```

```
x_train_1h[1:10, 1:10]
as.matrix(x_train_1h)[1:10, 1:10]
```

```
object.size(x_train_1h)/1024^2
object.size(as.matrix(x_train_1h))/1024^2
```

```
set.seed(3)
ival <- sample(seq(along = y_train), 2000)
accuracy <- function(pred, truth) {
            mean(drop(pred)==drop(truth))
}
```

```
library(glmnet)
fitlm <- glmnet(x_train_1h[-ival, ], y_train[-ival],
                family="binomial", standardize=FALSE)
pred.val <- predict(fitlm, x_train_1h[ival, ]) > 0
acc.val <- apply(pred.val, 2, accuracy, y_train[ival] > 0)
pred.test <- predict(fitlm, x_test_1h) > 0
acc.test <- apply(pred.test, 2, accuracy, y_test > 0)
```

```
dev.off()
plot(-log(fitlm$lambda), acc.val, type="l", col=2, ylim=c(0.4,1))
lines(-log(fitlm$lambda), acc.test, type="l", col=4)
legend("topleft", c("validation", "test"), lty=1, col=c(2,4))
abline(h=max(acc.test), lty=2, col=4)
max(acc.test)
```

```
model <- keras_model_sequential() %>%
        layer_dense(units=16, activation="relu",
                    input_shape=10000) %>%
        layer_dense(units=16, activation="relu") %>%
        layer_dense(units=1, activation="sigmoid")
```

```
compile(model, optimizer="rmsprop", metrics="accuracy",
        loss="binary_crossentropy")
```

```
hist.val <- fit(model, x_train_1h[-ival, ], y_train[-ival],
                epochs=20, batch_size = 512,
                validation_data=
                list(x_train_1h[ival,], y_train[ival]))
plot(hist.val)
```

```
hist.test <- fit(model, x_train_1h[-ival, ], y_train[-ival],
                epochs = 20, batch_size = 512,
                validation_data=list(x_test_1h, y_test))
plot(hist.test)
max(hist.test$metrics$val_accuracy)
```

Recurrent Neural Networks

- Many data sources are sequential in nature. Examples are
  - Documents such as book and movie reviews, newspaper articles, and tweets.
  - Time series of temperature, rainfall, wind speed, air quality and so on
  - Financial time series, where we track market indices, trading volumes, stock and bond prices, and exchange rates.
  - Recorded speech, musical recordings, and other sound recordings
- In a recurrent neural network (RNN), the input object

$$\boldsymbol{X} = \{X_1, X_2, \ldots, X_L\}$$

  is a sequence. e.g, a sequence of L words.
- RNNs are designed to accommodate and take advantage of the sequential nature of such input objects, while CNNs accommodate the spatial structure of image inputs.

Recurrent Neural Networks

- Suppose that
  - The input sequence: $X_l = (X_{l1}, X_{l2}, \ldots, X_{lp})^{\mathrm{T}}$
  - The K unit hidden layer: $A_l = (A_{l1}, A_{l2}, \ldots, A_{lK})^{\mathrm{T}}$
- A $K \times (p+1)$ matrix $\boldsymbol{W} = \{w_{kj}\}$ for input layer, a $K \times K$ matrix $\boldsymbol{U} = \{u_{ks}\}$ for the hidden-to-hidden layers, and a $K+1$ vector $\boldsymbol{B} = \{\beta_j\}$ are used for the weights. Then,

$$A_{lk} = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_l j + \sum_{s=1}^{K} u_{ks} A_{l-1,s}\right),$$

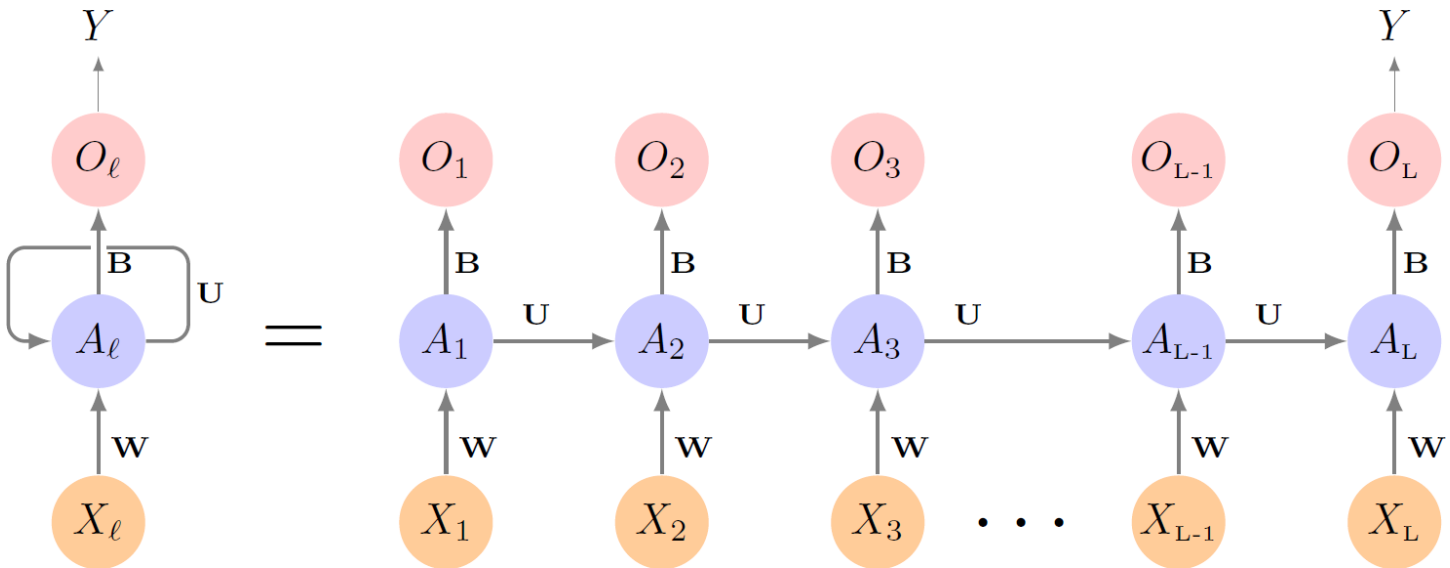
and the output O_l is computed as

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk}$$

for a quantitative response.

Recurrent Neural Networks

- A schematic of a simple recurrent neural network

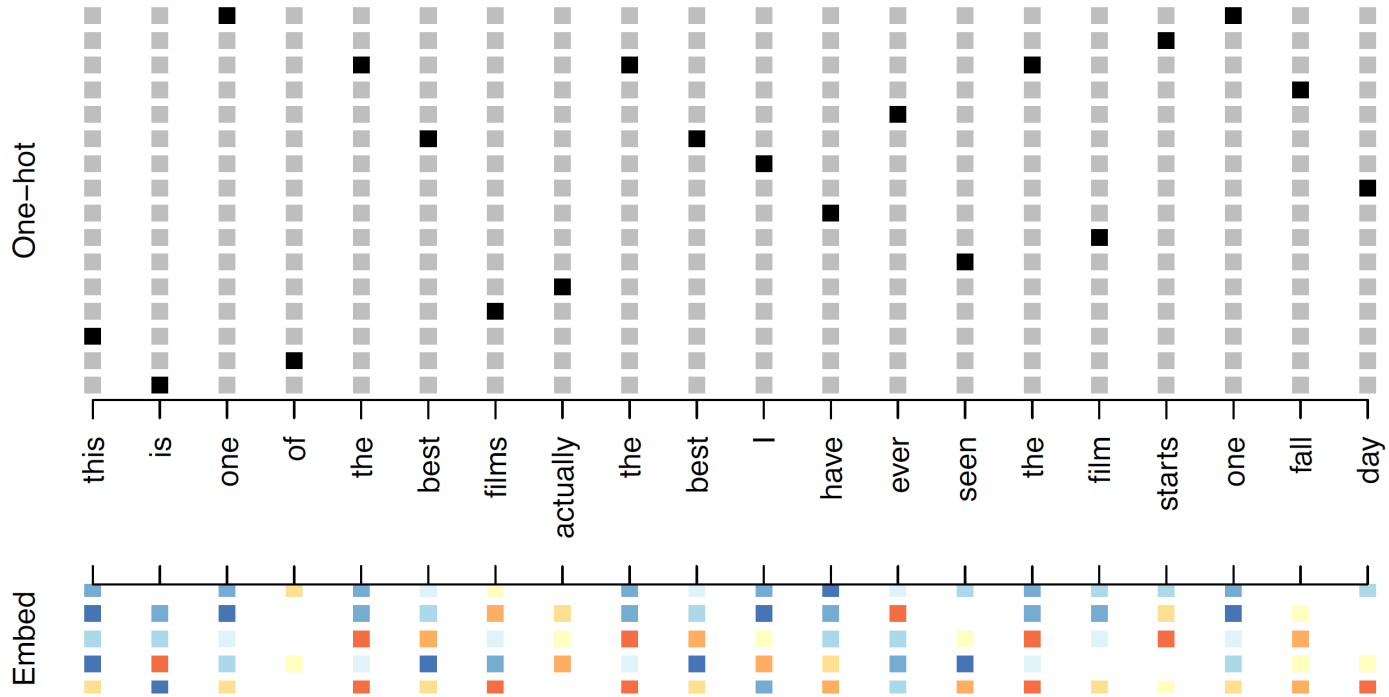


- The same wights $\boldsymbol{W}$, $\boldsymbol{U}$ and $\boldsymbol{B}$ are used as we process each element in the sequence — weight sharing.
- The final output O_L is only used, i.e., $O_1, O_2, \ldots, O_{L-1}$ are not used.

Sequential Models for Document Classification

- Previously, we used the bag-of-words model for the `IMDb` data. Now, we consider the sequence of words occurring in a document to make predictions.
- However, there is a dimensionality problem– each word in our document is represented by a one-hot-encoded vector (dummy variable) with 10,000 elements.
- A popular approach is a much lower-dimensional embedding space, which is based on a set of m real numbers, and none of which are typically zero.
- We need a matrix $\boldsymbol{E}$ of dimension $m \times 10,000$.
  - Each column is indexed by one of the 10,000 words.
  - The values in that column give the m coordinates for that word in the embedding space.

Sequential Models for Document Classification



- If we have a large corpus of labeled documents, we can have the neural network learn $\boldsymbol{E}$ as part of the optimization – referred to as an embedding layer.
- A pre-computed matrix $\boldsymbol{E}$ can be inserted into the embedding layer – weight freezing.

Sequential Models for Document Classification

- Each document is now represented as a sequence of m vectors representing the sequence of words.
- Next, each document is limited to the last L words, so $X = \{X_1, X_2, \ldots, X_L\}$, where X_l in the sequence has m components.
- Finally, we apply RNN the training documents of length L. If there are K hidden units,
  - $\boldsymbol{W}$ has $K \times (m + 1)$ parameters.
  - $\boldsymbol{U}$ has $K \times K$ parameters.
  - $\boldsymbol{B}$ has $2(K + 1)$ parameters for the two-class logistic regression.
- If the embedding layer $\boldsymbol{E}$ is learned, we have additional $m \times D$ parameters, e.g., $D = 10,000$ in our example.
- The pre-trained embedding matrix $\boldsymbol{E}$ can be employed by `Glove` and `wird2vec`.

Example: IMDb Data

- We first calculate the lengths of the documents.
- We see that over 91% of the documents have fewer than 500 words.

```
wc <- sapply(x_train , length)
summary(wc)
sum(wc <= 500) / length(wc)
```

- RNN requires all the document sequences to have the same length, so we restrict $L = 500$.
- If a document has shorter than 500, we get padded with zeros upfront.

```
maxlen <- 500
x_train <- pad_sequences(x_train , maxlen = maxlen)
x_test <- pad_sequences(x_test , maxlen = maxlen)
rbind(dim(x_train), dim(x_test))
```

```
model <- keras_model_sequential() %>%
         layer_embedding(input_dim=10000, output_dim=32) %>%
         layer_lstm(units=32) %>%
         layer_dense(units=1, activation="sigmoid")
```

```
compile(model, optimizer = "rmsprop",
        loss="binary_crossentropy", metrics=c("acc"))
```

```
#### Do not run ##########################################
#### It takes a few minutes ##############################
history <- fit(model, x_train, y_train, epochs=10,
               batch_size=128,
               validation_data=list(x_test, y_test))
plot(history)
```

```
pred <- predict(model, x_test) > 0.5
mean(abs(y_test==as.numeric(pred)))
```

When to Use Deep Learning

- We see daily reports of new success stories for deep learning.
- The CNNs have really revolutionized image classification. Also, there are numerous successes of RNNs in speech and language translation, forecasting, and document modeling.
- A big question:
  - Should we discard all our older tools, and use deep learning on every problem with data?
- The Occam's razor principle: when faced with several methods that give roughly equivalent performance, pick the simplest.
- If the simpler tools perform well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches.
- Typically, we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority.