Neural Networks Review

Arin Ghazarian
Chapman University

Kullback-Leibler (KL) divergence

- If we have two separate probability distributions P (x) and Q (x) over the same random variable x, we can measure how different these two distributions are using the Kullback-Leibler (KL) divergence:

$$D_{\mathrm{KL}}(P\|Q) = \mathbb{E}_{\mathrm{x}\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{\mathrm{x}\sim P}\left[\log P(x) - \log Q(x)\right].$$

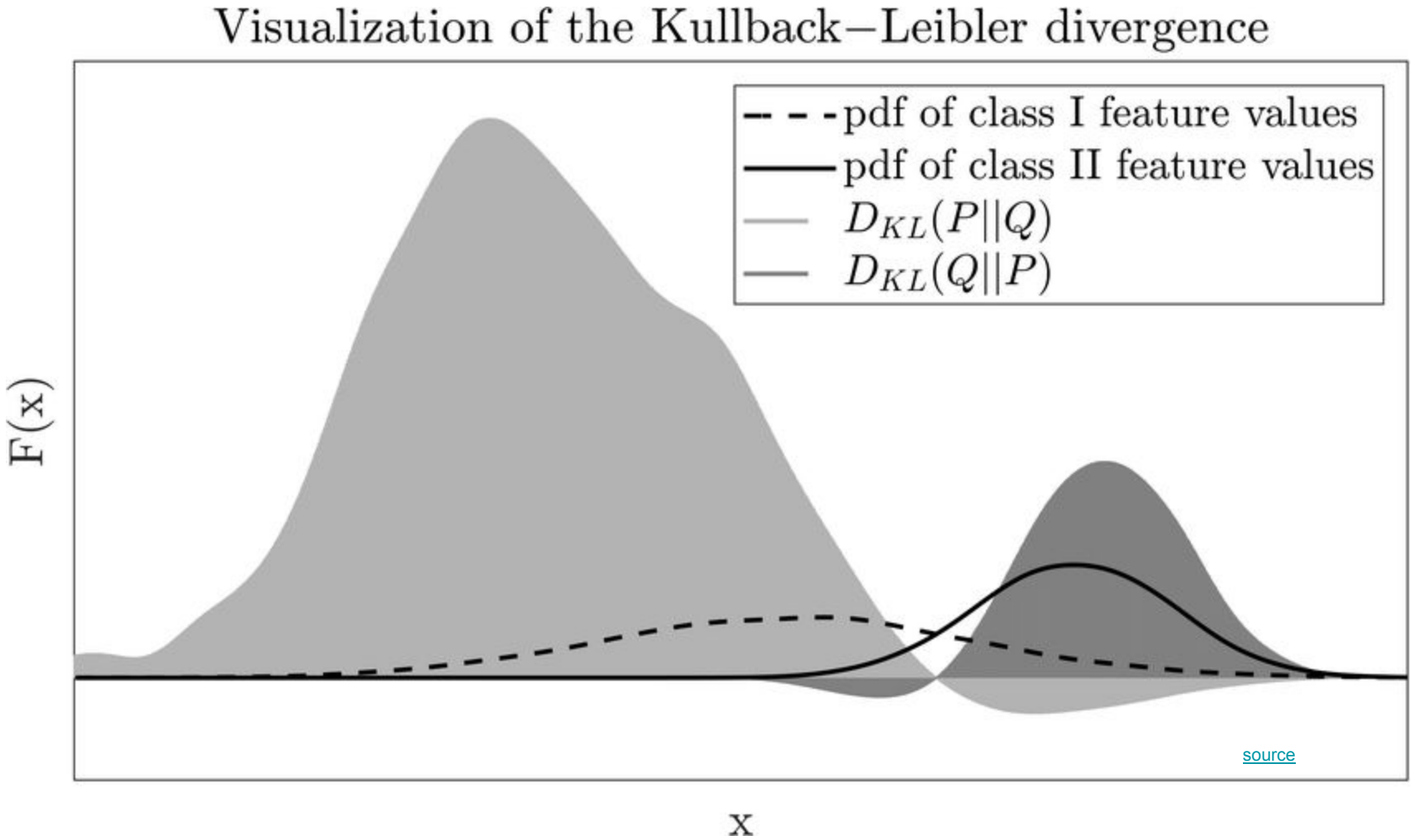
Kullback-Leibler (KL) divergence

- In the case of discrete variables, it is the extra amount of information (measured in bits if we use the base 2 logarithm, but in machine learning we usually use nats and the natural logarithm) needed to send a message containing symbols drawn from probability distribution P, when we use a code that was designed to minimize the length of messages drawn from probability distribution Q.

Kullback-Leibler (KL) divergence as a Measure of Distance

- The KL divergence has many useful properties, most notably that it is nonnegative.
- The KL divergence is 0 if and only if P and Q are the same distribution in the case of discrete variables, or equal “almost everywhere” in the case of continuous variables.
- Because the KL divergence is non-negative and measures the difference between two distributions, it is often conceptualized as measuring some sort of distance between these distributions.
- However, it is not a true distance measure because it is not symmetric:

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P) \text{ for some } P \text{ and } Q$$


Visualization of the Kullback–Leibler divergence
pdf of class I feature values
pdf of class II feature values
$D_{KL}(P||Q)$
$D_{KL}(Q||P)$
F(x)
x
source

Cross-entropy

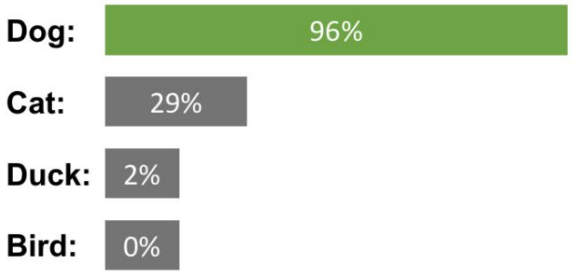
- A quantity that is closely related to the KL divergence is the cross-entropy $H(P,Q) = H(P) + D_{KL}(PQ)$, which is similar to the KL divergence but lacking the term on the left:

$$H(P, Q) = -\mathbb{E}_{\mathrm{x} \sim P} \log Q(x).$$

- Minimizing the cross-entropy with respect to Q is equivalent to minimizing the KL divergence, because Q does not participate in the omitted term.

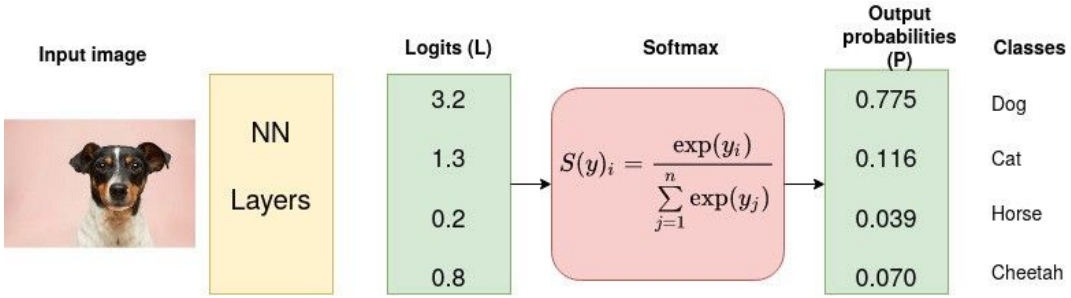
Likelihood Example

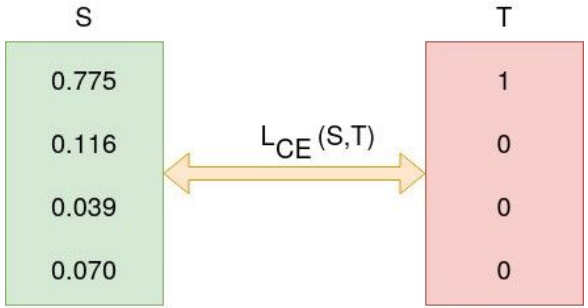
source



$L = .96^{1} \times .29^{0} \times .02^{0} \times .0^{0} = 0.96$

Cross Entropy Data Distribution vs Model Distribution





https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

LogLikelihood

$$loglik(\theta) = \sum_{j=1}^{M} y_j log(\overline{y_j})$$

It is easier to minimise the negative log-likelihood function than maximising the direct likelihood

$$loglik(\theta) = -\sum_{j=1}^{M} y_j log(\overline{y_j})$$

Cross-entropy and Loglikelihood

- For a multiclass classification problem, we use cross-entropy as a loss function.
  - The average bits of information required to identify an event drawn from the estimated probability distribution f(x), rather than the true distribution g(x)

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x).$$

P=y_i and Q=$\bar{y}_i$

H(P,Q)=-$loglik(\theta) = -\sum_{j=1}^{M} y_j log(\overline{y_j})$

Thus cross-entropy is equal to negative log-likelihood. By minimizing the negative loglikelihood, in fact we are maximizing the likelihood

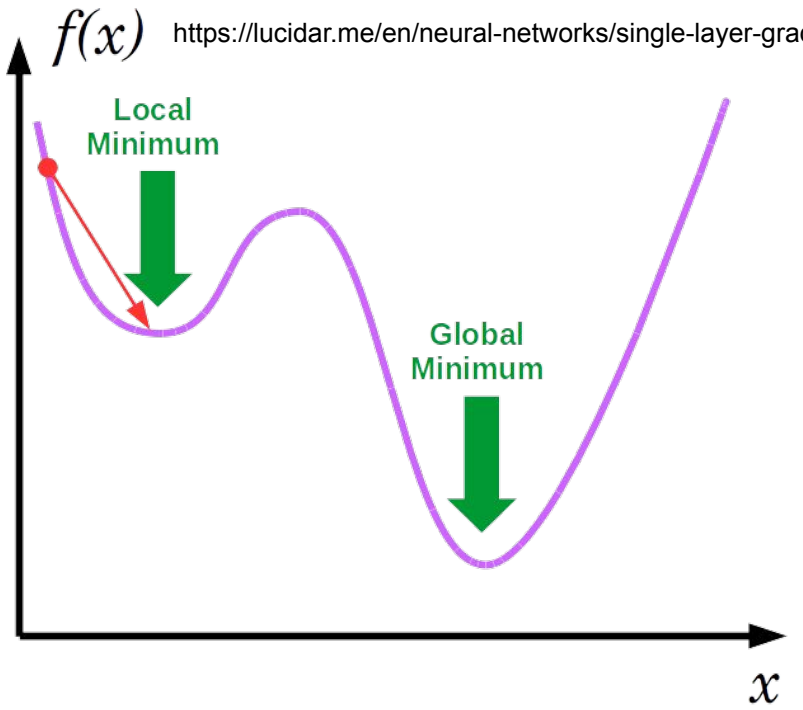
Gradient descent

- Hiker in the fog approach
- Imagine you are hiking in the fog and need to get down a mountain.
- You cannot see where the optimal downhill path it.
- But you can see your immediate surroundings.
- So you decide to choose the steepest downhill path at any point.
- If you are lucky, this method will get you down the hill. If not, you might get stuck in some valley or ravine, still up the mountain.

source

Gradient descent

- Taking the best possible downhill path at any point is the essence of the gradient descent algorithm.



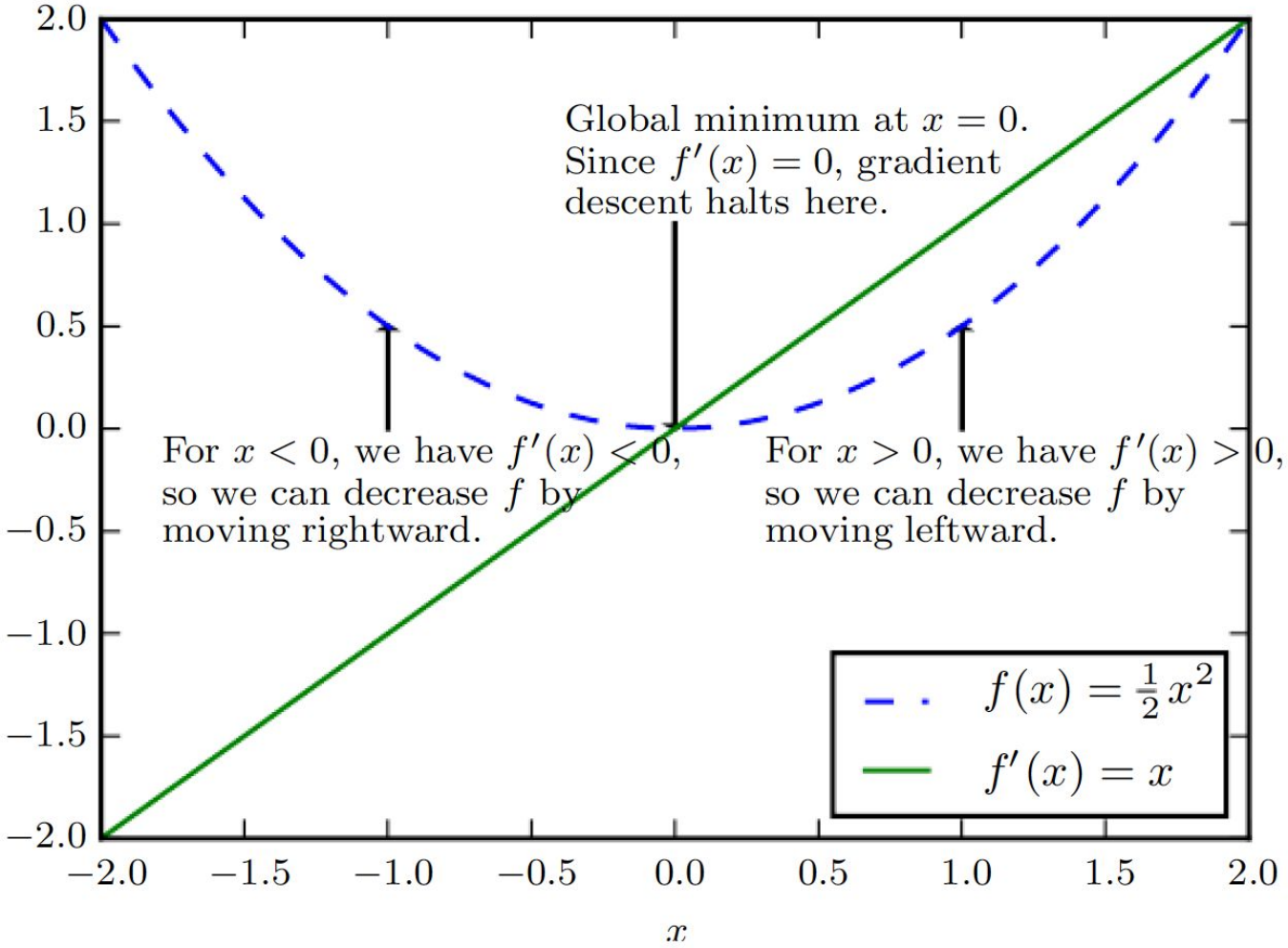
Derivative

- Using the derivative we can estimate how a small change in input will scale the output:

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

Gradient Descent

- In gradient descent, we move in small steps **in the opposite to the derivative sign**, hoping it will reduce the error function.
- The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y.
- For example, we know that $f(x - \varepsilon \operatorname{sign}(f'(x)))$ is less than $f(x)$ for small enough ε. We can thus reduce f (x) by moving x in small steps with opposite sign of the derivative. This technique is called gradient descent (Cauchy, 1847).


Global minimum at $x = 0$.
Since $f'(x) = 0$, gradient
descent halts here.
For $x < 0$, we have $f'(x) < 0$,
so we can decrease f by
moving rightward.
For $x > 0$, we have $f'(x) > 0$,
so we can decrease f by
moving leftward.
$f(x) = \frac{1}{2}x^2$
$f'(x) = x$
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x

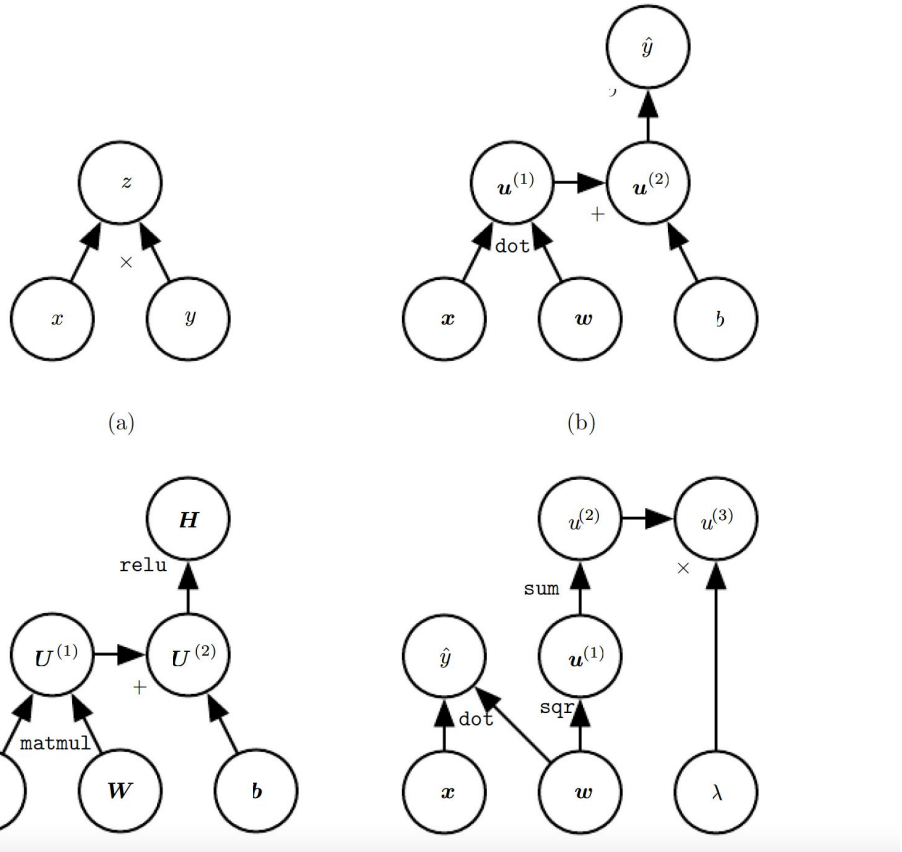
Gradient Descent

- Gradient descent proposes a new point, where epsilon is the learning rate, a positive scalar determining the size of the step:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

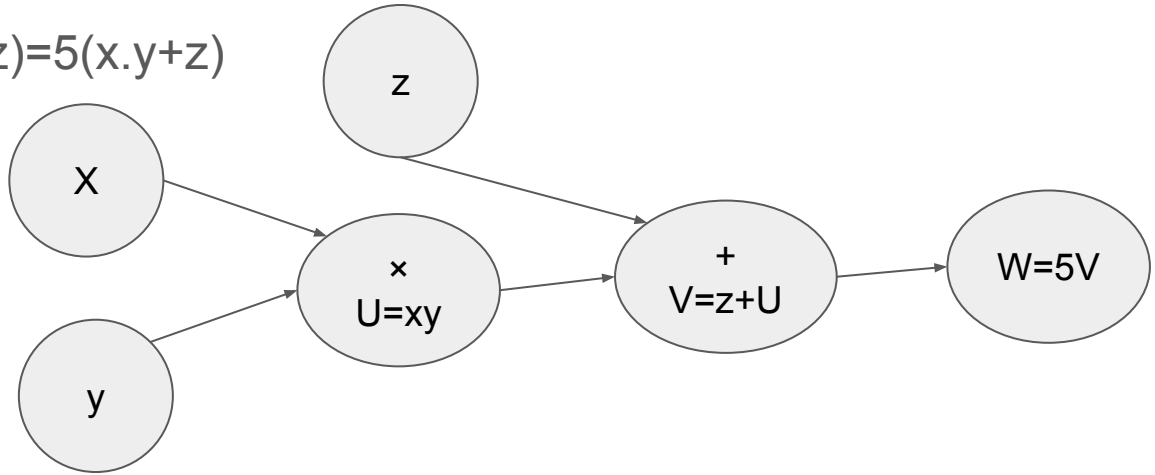
- Gradient descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm, and just jump directly to the critical point by solving the equation $\nabla_x f(x) = 0$ for x.

Computational Graphs

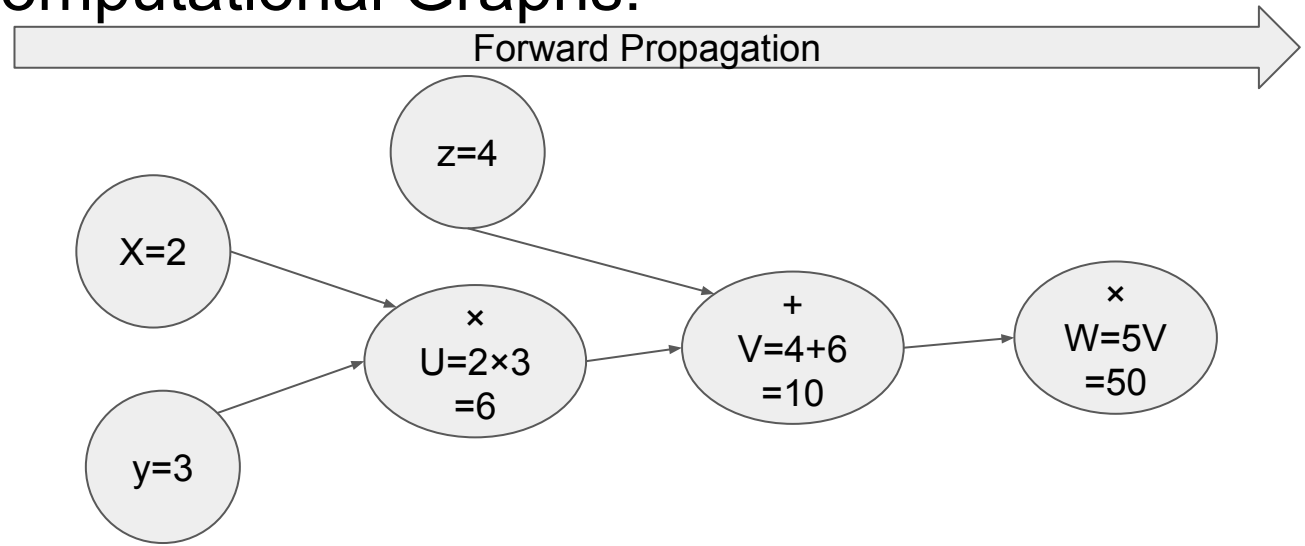


Computational Graphs

f(x,y,z)=5(x.y+z)

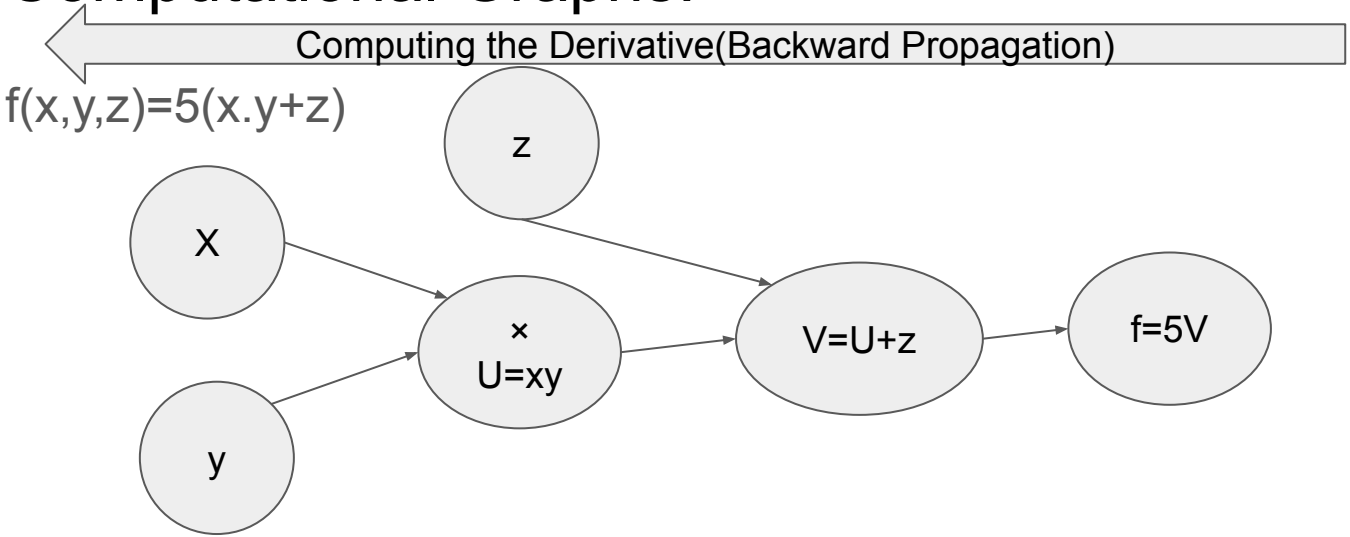


Computational Graphs:



x=2, y=3, z=4

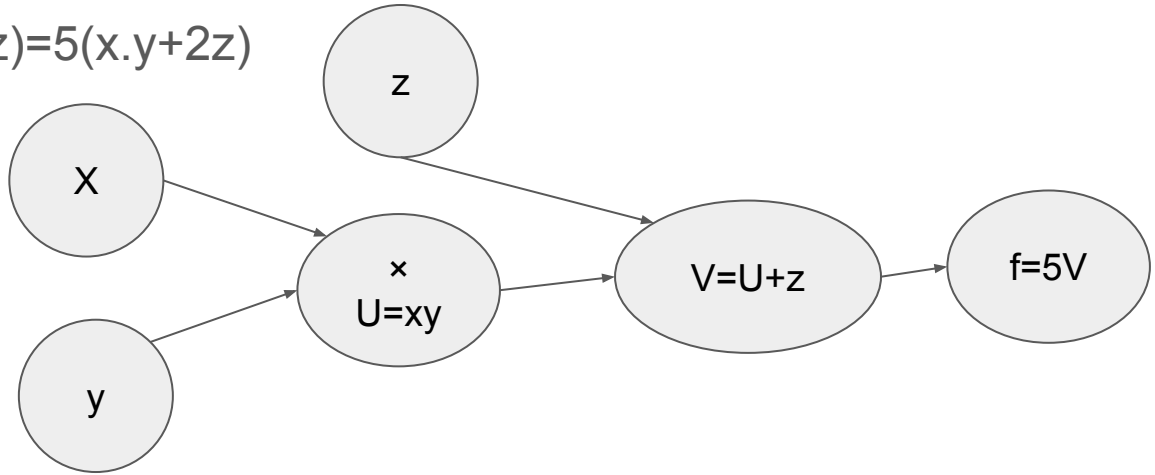
Computational Graphs:



$\partial f/\partial V=5$
$\partial V/\partial z=1$
$\partial f/\partial z=\partial f/\partial V.\partial V/\partial z=5 \ . \ 1=5$

Computational Graphs: Computing the Derivative

f(x,y,z)=5(x.y+2z)

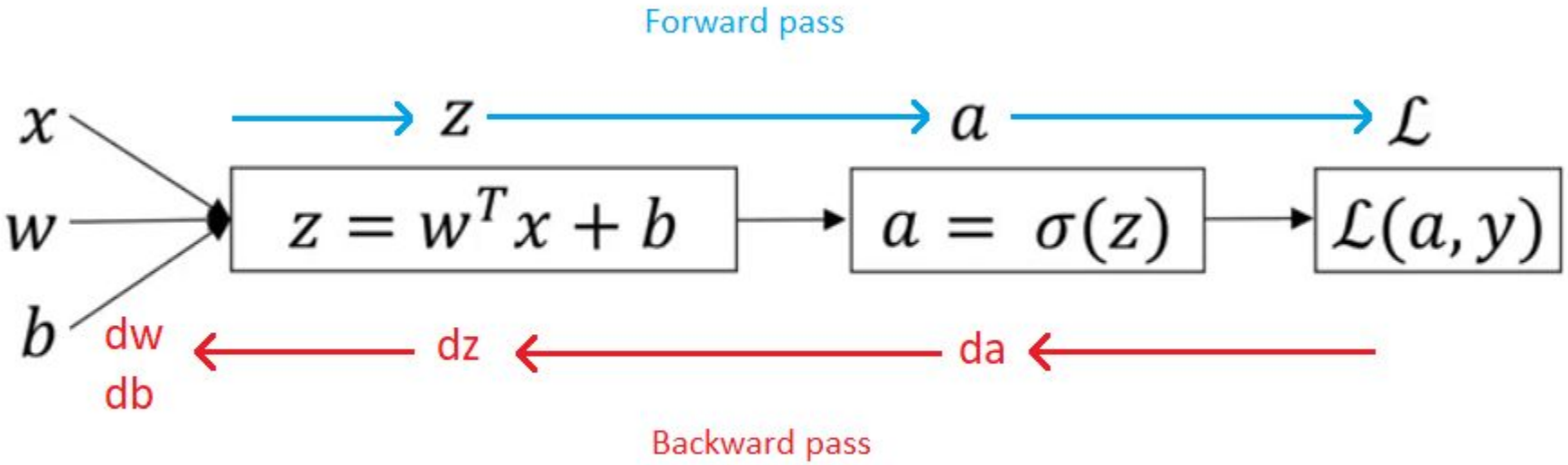


x=2, y=3
z=4→4.01
V=10→10.01
f=50→50.05

∂f/∂V=5
∂V/∂z=1
∂f/∂z=∂f/∂V.∂V/∂z=5 . 2=10

Computational Graph For Logistic Regression

da means ∂L/∂a and dz means ∂L/∂z



https://datahacker.rs/gradient-descent-neural-networks/

Derivation Rules: Reminder

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

$$\sigma'(x) = \frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$

$$(\ln x)' = \frac{1}{x}$$

$$(f(x)\,g(x)\,)' = f'(x)\,g(x) + f(x)\,g'(x)$$

$$\frac{d}{dx}[\ln u] = \frac{1}{u}\cdot u' = \frac{u'}{u}$$

L(a,y)=-[ylog(a)+(1-y)log(1-a)]

∂L/∂a=da=-[(y.(1/a))+((1-y).(-1/(1-a)))]= -[((1-a)y+a(y-1))/(a(1-a))]=-(y-a)/(a(1-a))

$$\frac{dL}{dz} = \frac{dL}{da} \times \frac{da}{dz}$$

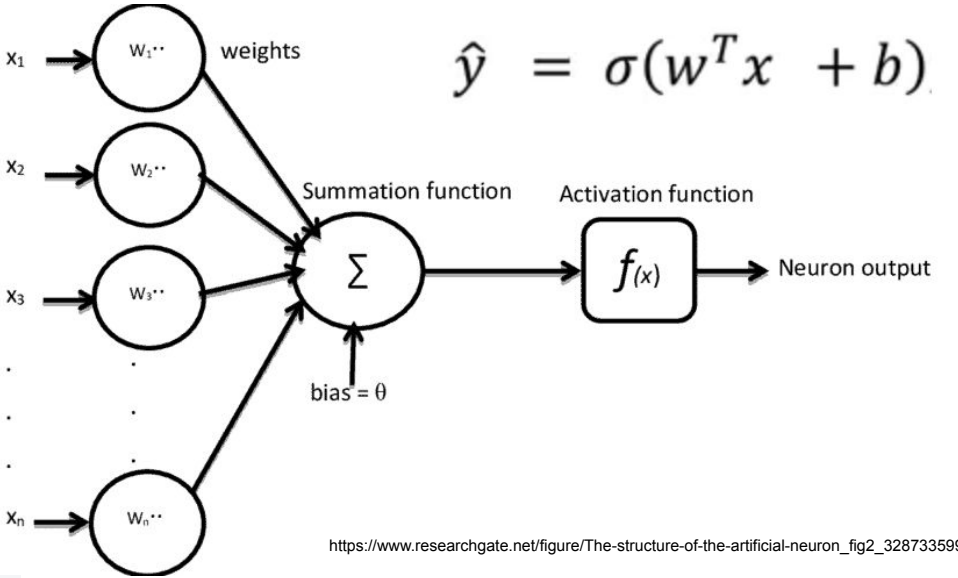
$$\frac{dL}{dz} = \frac{a-y}{a(1-a)} \times a(1-a) = a - y$$

∂L/∂W=∂L/∂z. ∂z/∂w=(a-y)x

∂L/∂b=∂L/∂z. ∂z/∂b=(a-y)

Logistic Regression as a Single Neuron Neural Network

- Logistic regression is in fact a single neuron neural network model with a sigmoid activation function: $P(X)=\sigma(\beta_0+\beta_1 x)$
- The cost function is the cross entropy between the predictions and actuals
- We used gradient descent to find the W and b which minimizes the cost



https://www.researchgate.net/figure/The-structure-of-the-artificial-neuron_fig2_328733599

$$P = \frac{1}{1+e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+\beta_3 X_3+\ldots+\beta_n X_n)}}$$

Vectorizing Logistic Regression

- Assuming we have M training samples, we need to go over each sample in a loop

$J = 0, dw_1 = 0, dw_2 = 0, db = 0$

for i = 1 to m:

$$z^{(i)} = w^T x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

$$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$

$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

$$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$

$$db \mathrel{+}= dz^{(i)}$$

$J = J/m, dw_1 = dw_1/m, dw_2 = dw_2/m$

$db = db/m$

https://www.codingninjas.com/codestudio/library/vectorizing-logistic-regression

Vectorizing Logistic Regression:Forward Propagation

- The size of training matrix X will be (N_x by M). $X = [x^{(1)}\ x^{(2)}\ x^{(3)}\ x^{(4)} \dots x^{(m)}]$ (n_x, m) $\mathbb{R}^{n_x \times m}$
  - N_x is the number of features
- W is (N_x by 1) matrix (W of the logistic regression model)

$$Z = [z^{(1)}\ z^{(2)}\ z^{(3)} \dots z^{(m)}]$$
$$= w^T X + [b\ b\ b \dots b]$$
$$= w^T [x^{(1)}\ x^{(2)}\ x^{(3)} \dots x^{(m)}] + [b\ b \dots b]$$
$$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \dots \quad w^T x^{(m)} + b]$$

$w^T x^{(1)} + b = z^{(1)}$, $w^T x^{(2)} + b = z^{(2)}$, $w^T x^{(m)} + b = z^{(m)}$

$(1 \times m)$ dimensions

$(1 \times m)$ dimensions

$$A = [a^{(1)}\ a^{(2)}\ a^{(3)} \dots a^{(m)}] = \sigma(Z)$$

https://www.codingninjas.com/codestudio/library/vectorizing-logistic-regression

Vectorizing Logistic Regression:Backward Propagation

$$db = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)}$$

$$= \frac{1}{m}\, np.sum(dz)$$

$$dw = \frac{1}{m} X\, dz^{T}$$

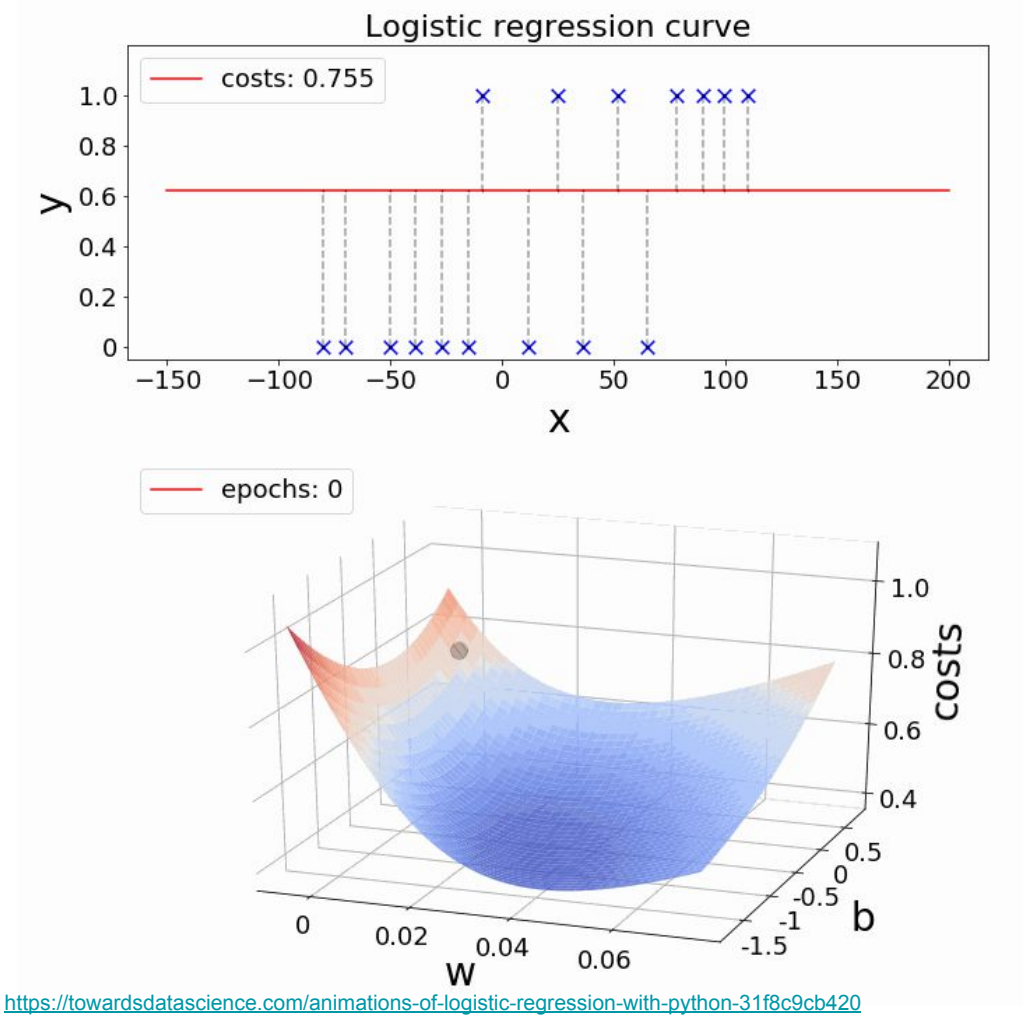
$$= \frac{1}{m}\begin{bmatrix} | & | & | & & | \\ x^{(1)} & x^{(2)} & x^{(3)} & \cdots & x^{(m)} \\ | & | & | & & | \end{bmatrix}_{(n\times m)} \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \\ dz^{(m)} \end{bmatrix}_{(m\times 1)}$$

$$= \frac{1}{m}\left[x^{(1)}dz^{(1)} + x^{(2)}dz^{(2)} + \cdots + x^{(m)}dz^{(m)}\right]_{(n\times 1)}$$

$$w := w - \alpha\, dw$$

$$b := b - \alpha\, db$$

https://www.codingninjas.com/codestudio/library/vectorizing-logistic-regression



https://towardsdatascience.com/animations-of-logistic-regression-with-python-31f8c9cb420

Chain Rule of Calculus

- Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that y = g(x) and z = f (g(x)) = f (y). Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- We can generalize this beyond the scalar case. Suppose that $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to R . If y = g(x) and z = f(y), then

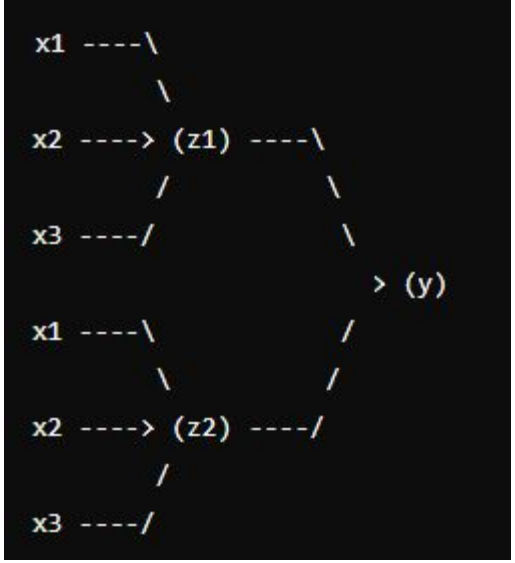
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}.$$

- In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

where ∂y/∂x is the n × m Jacobian matrix of g.

Jacobian-gradient Products



- The gradient of a variable x can be obtained by multiplying a Jacobian matrix $\partial y/\partial x$ by a gradient $\nabla_y z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.
- https://suzyahyah.github.io/calculus/machine%20learning/2018/04/04/Jacobian-and-Backpropagation.html
- Example: Given a fully connected neural net with 3 inputs, x_1 , x_2 , x_3 and hidden layer with two neurons z_1 , z_2 , and a single scalar output y:

Gradient

Jacobian

$$\frac{\delta y}{\delta x} = \frac{\delta y}{\delta z} \times \frac{\delta z}{\delta x} = \begin{bmatrix} \frac{\delta y}{\delta z_1} & \frac{\delta y}{\delta z_2} \end{bmatrix} \times \begin{bmatrix} \frac{\delta z_1}{\delta x_1} & \frac{\delta z_1}{\delta x_2} & \frac{\delta z_1}{\delta x_3} \\ \frac{\delta z_2}{\delta x_1} & \frac{\delta z_2}{\delta x_2} & \frac{\delta z_2}{\delta x_3} \end{bmatrix}$$

Demo: Training a Logistic Regression Using Gradient Descent

- http://localhost:8888/notebooks/jupyter-notebooks/chapman-generative-AI/NN-review-Logistic-regression.ipynb

Automatic Differentiation

- Automatic differentiation or autodiff is a set of techniques to evaluate the derivative of a function specified by a computer program.
- AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.).By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

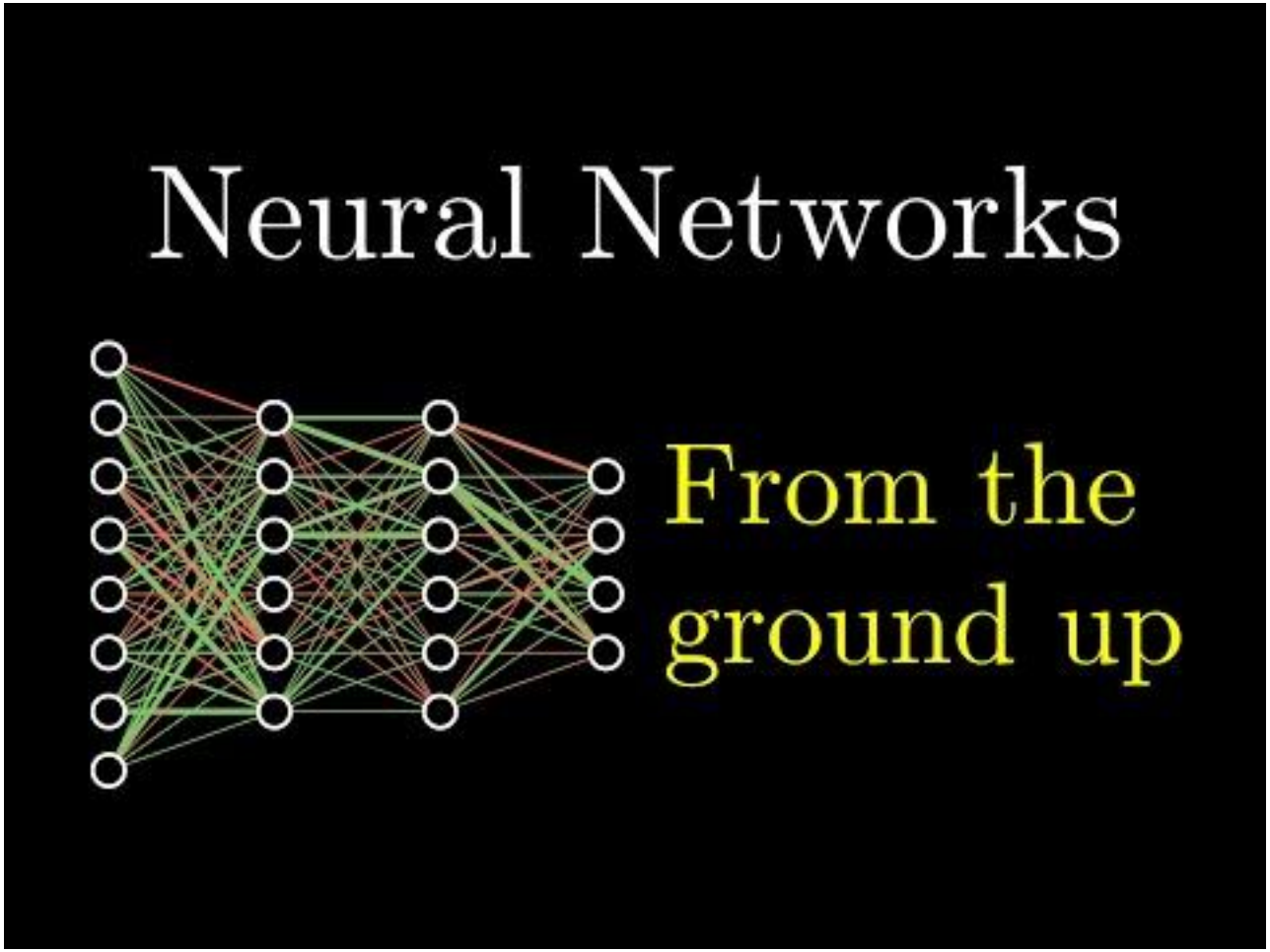
$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

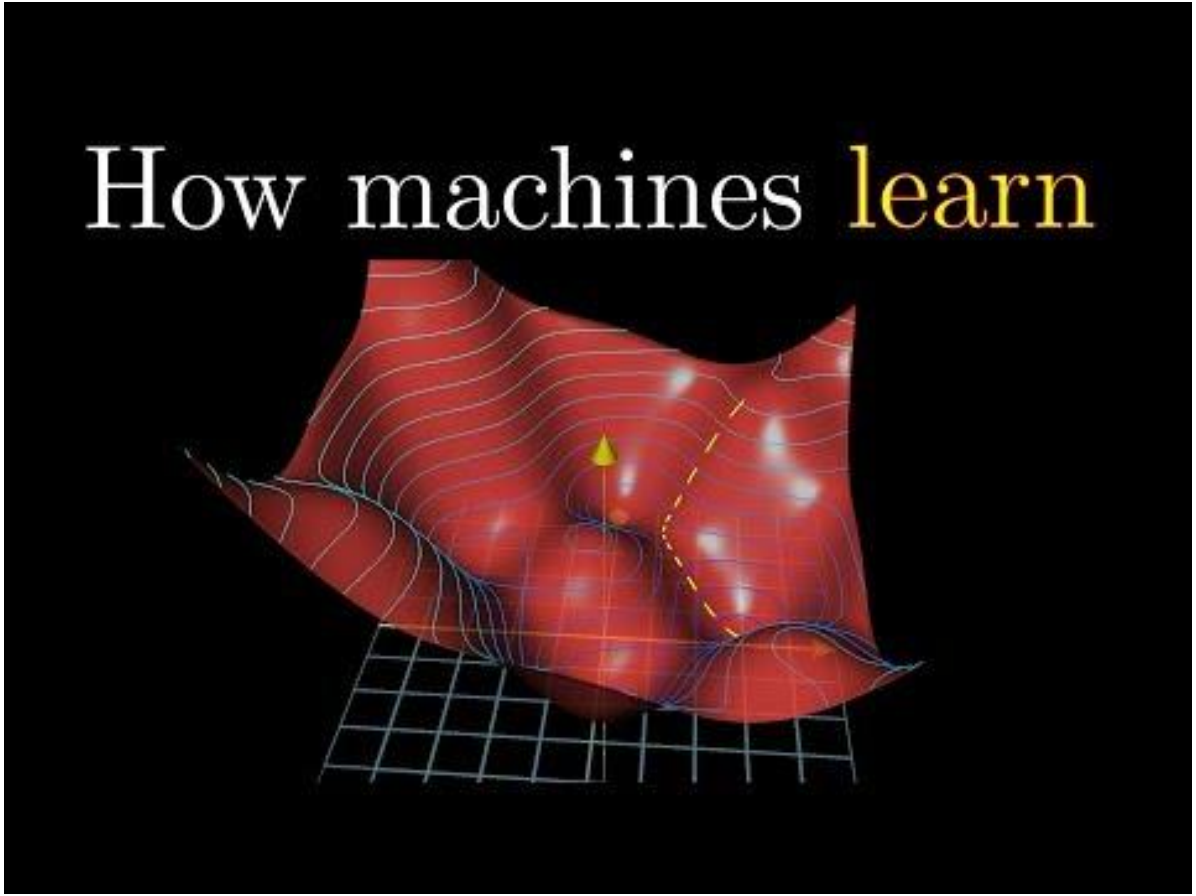
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

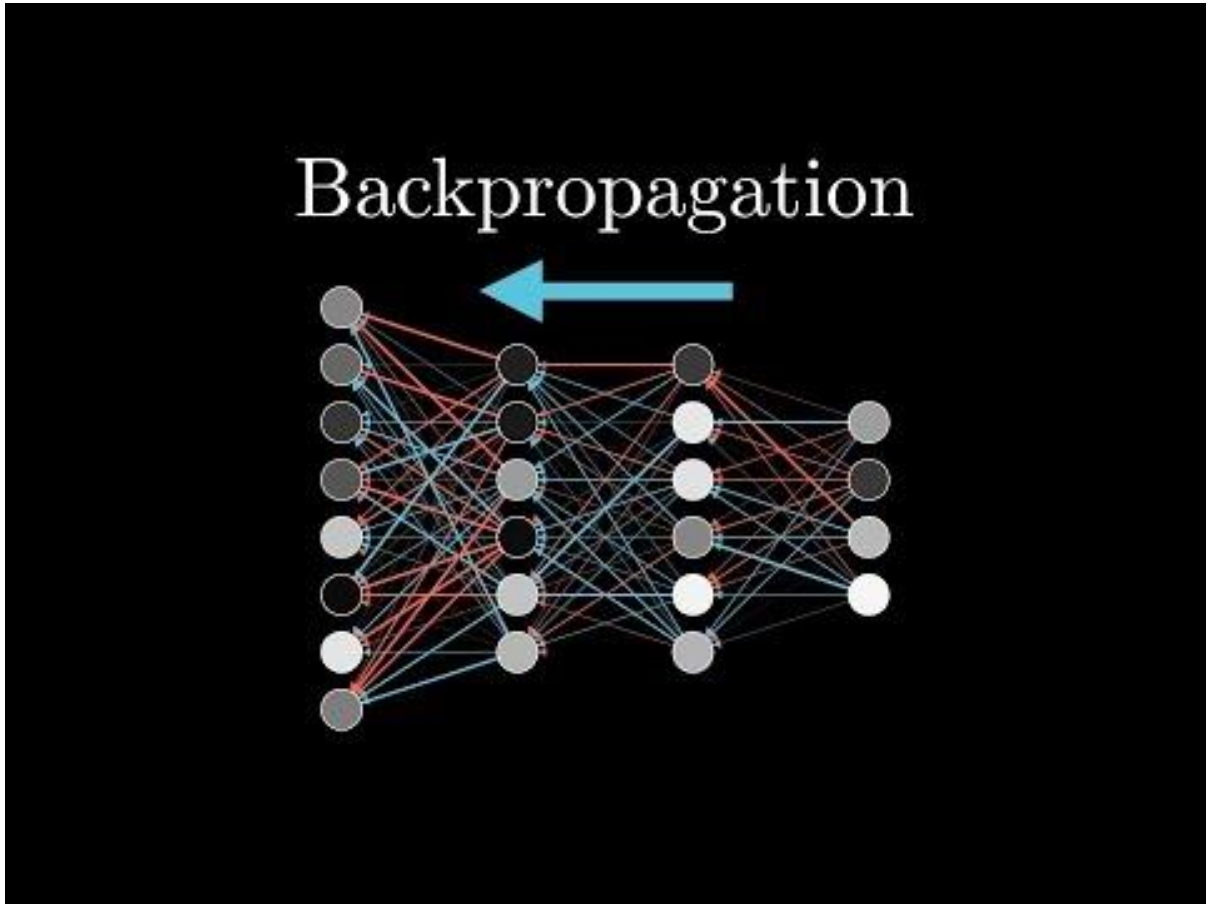
end for



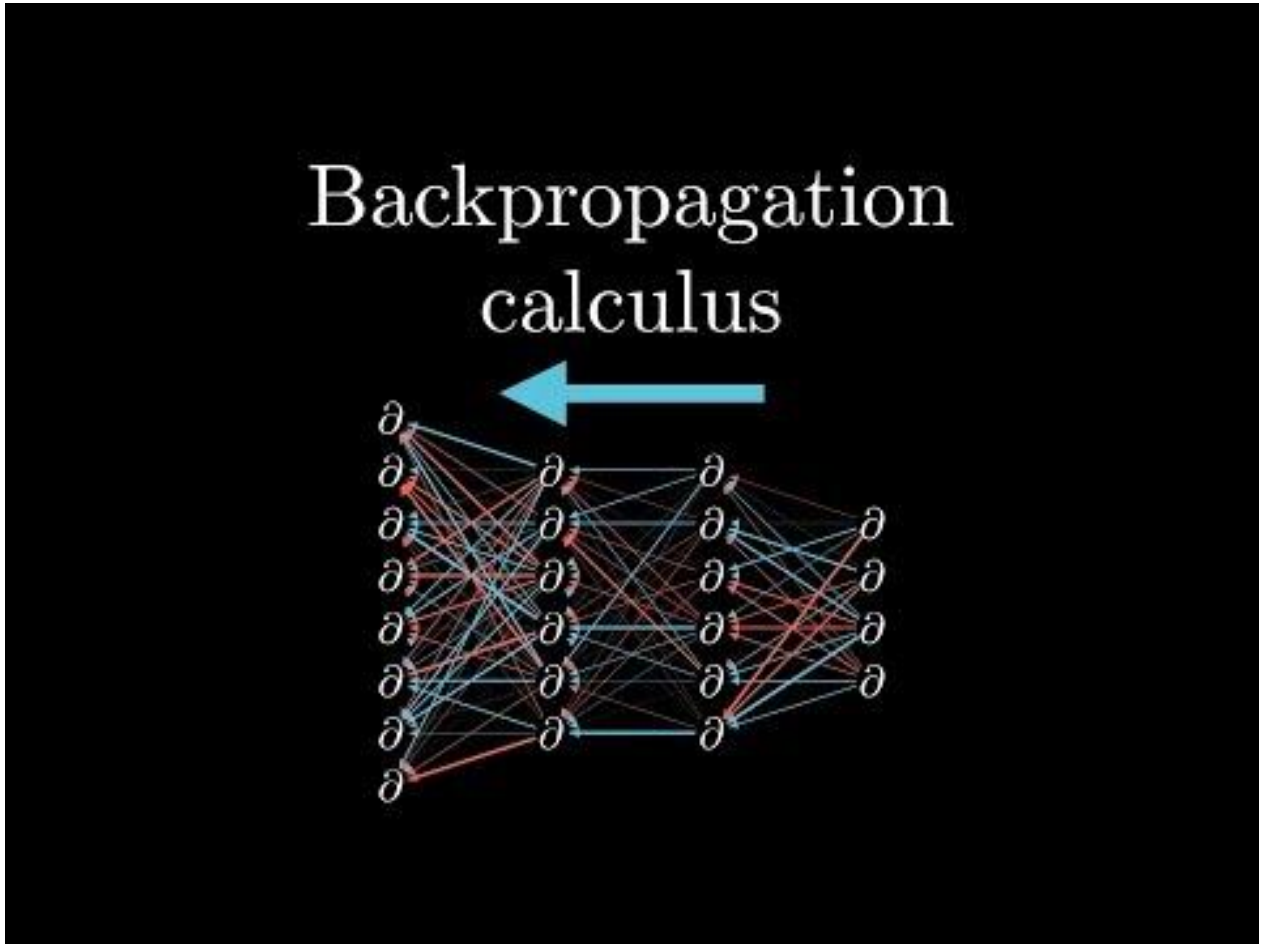
https://www.youtube.com/watch?v=aircAruvnKk



https://www.youtube.com/watch?v=IHZwWFHWa-w&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=2



https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3



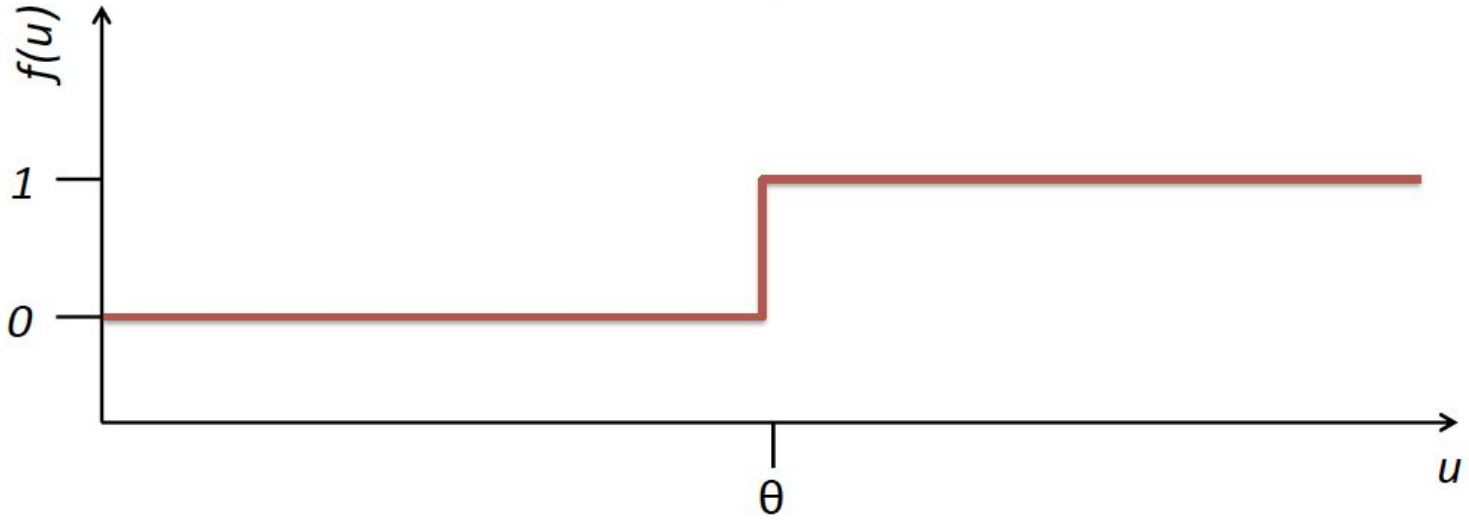
https://www.youtube.com/watch?v=tIeHLnjs5U8&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=4

Activation Functions

Step Function

Non-continuous function:

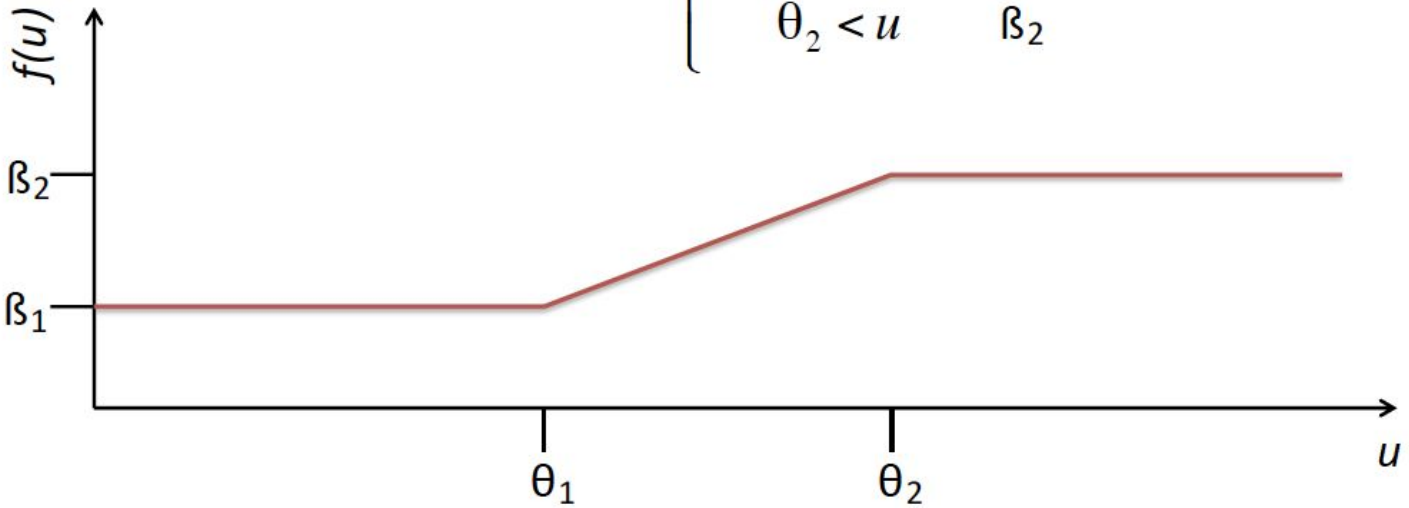
$$f(u) = \begin{cases} 0 & u < \theta \\ 1 & u \geq \theta \end{cases} \qquad f : \Re \rightarrow \{0,1\}$$



Rectified linear (ReLU)

Non-continuously differentiable function

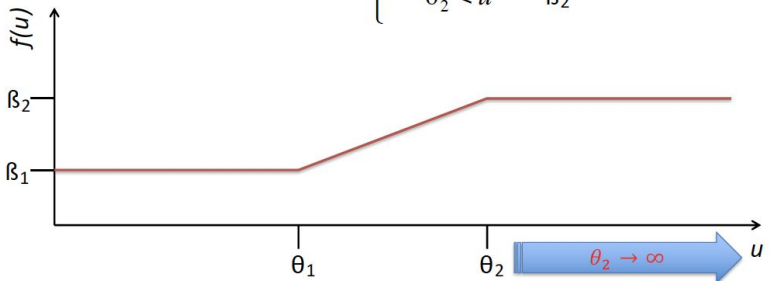
$$f(u) = \begin{cases} u < \theta_1 & ß_1 \\ \theta_1 < u < \theta_2 & u \\ \theta_2 < u & ß_2 \end{cases} \qquad f : \Re \rightarrow [\theta_1, \theta_2]$$



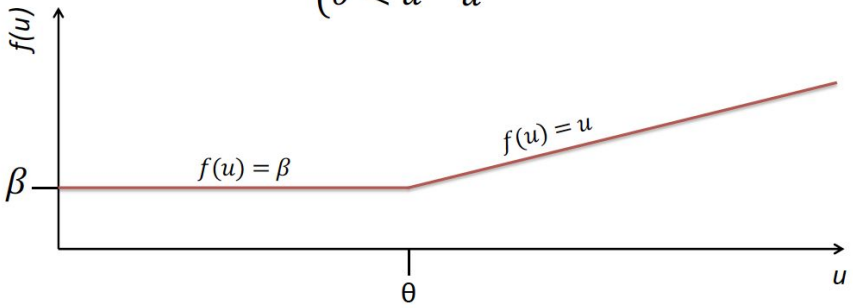
Rectified linear

Non-continuously differentiable functions

$$f(u) = \begin{cases} u < \theta_1 & ß_1 \\ \theta_1 < u < \theta_2 & u \\ \theta_2 < u & ß_2 \end{cases} \quad f : \Re \to [\theta_1, \theta_2]$$

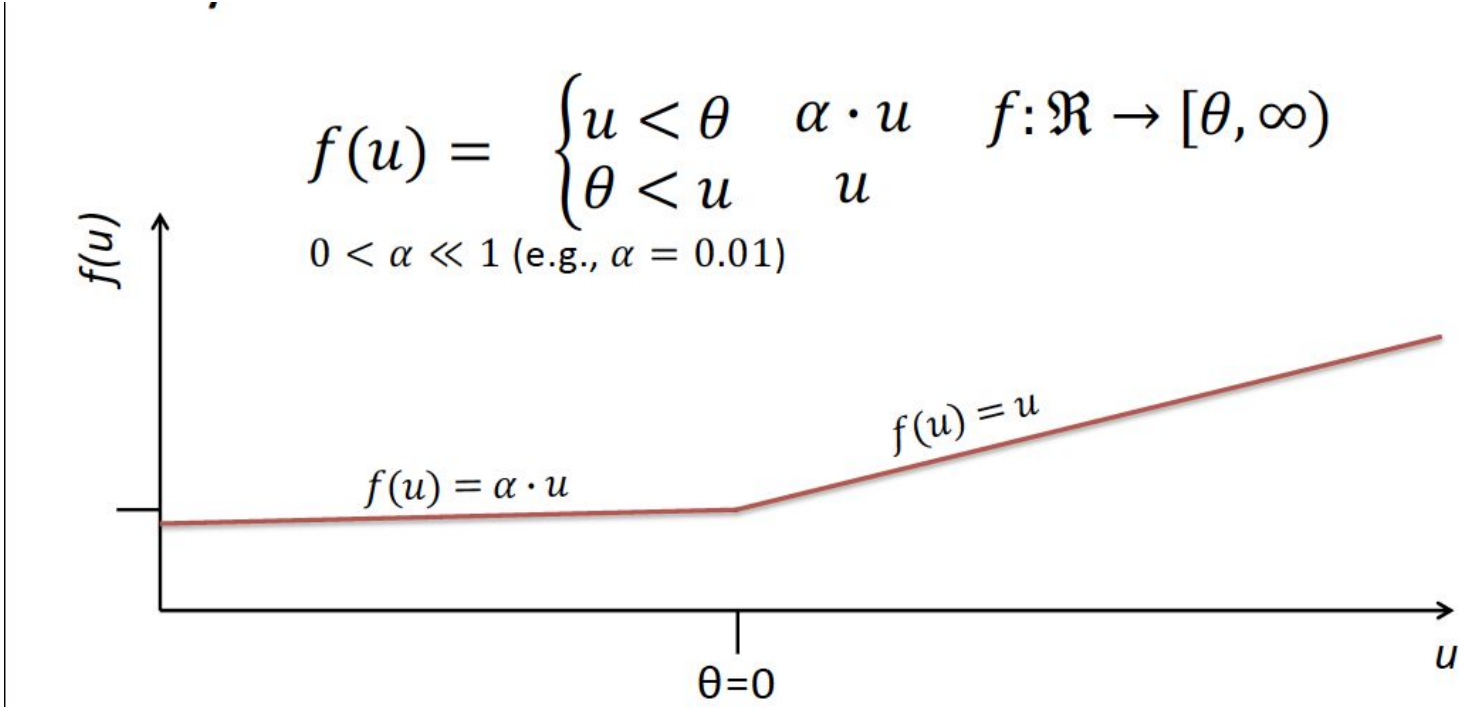


$$f(u) = \begin{cases} u < \theta & \beta \\ \theta < u & u \end{cases} \quad f : \Re \to [\theta, \infty)$$



Leaky rectified linear

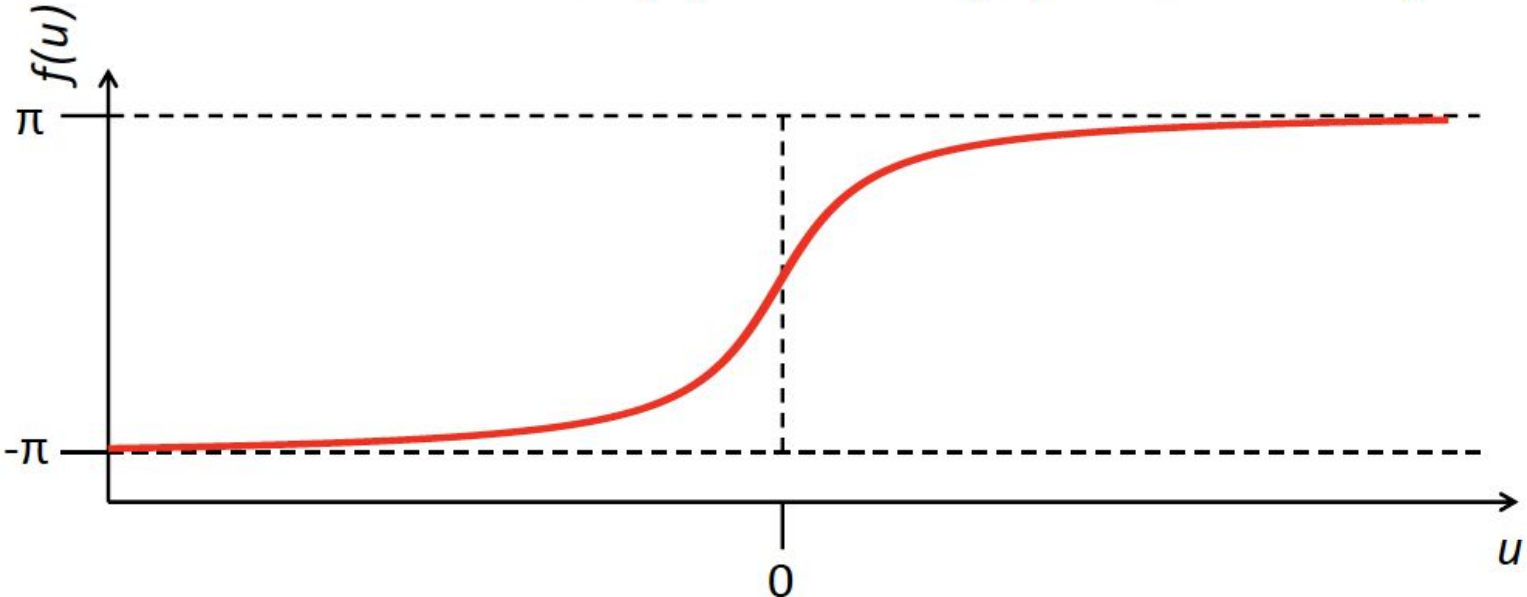
Non-continuously differentiable functions



Sigmoid functions: Arctan

continuously differentiable

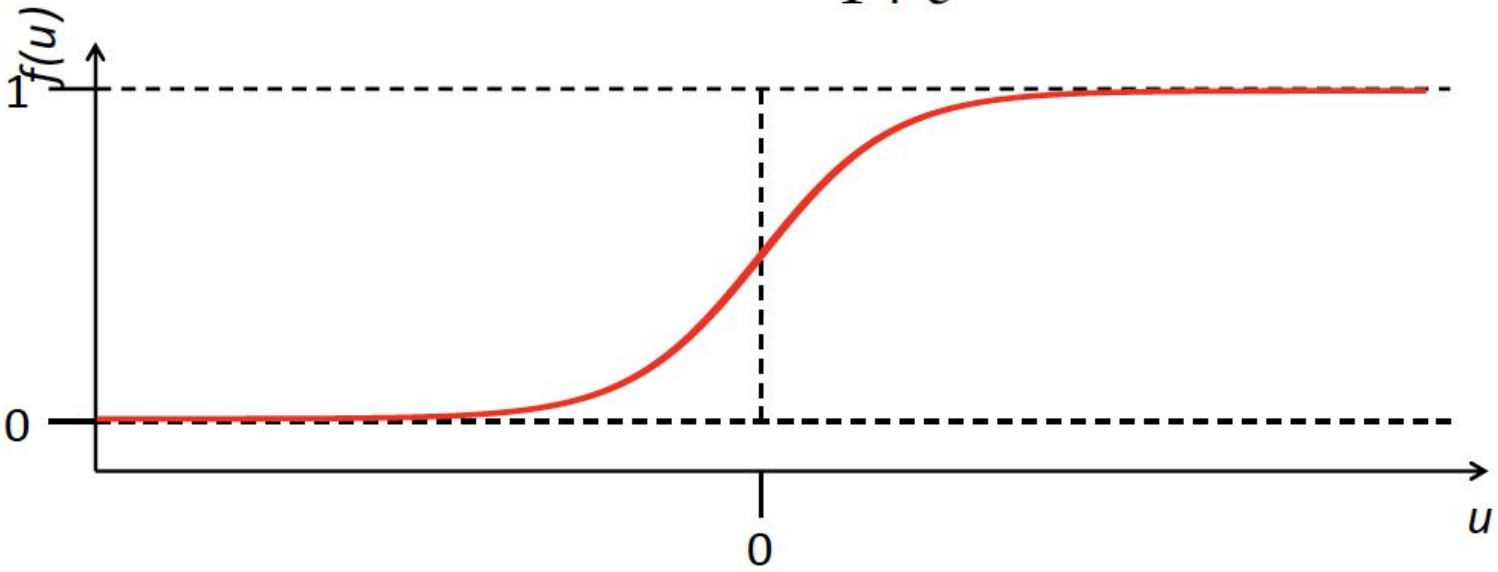
$$f(u) = \arctan(u) \qquad f : \Re \rightarrow [-\pi, \pi]$$



Sigmoid functions: Logistic function

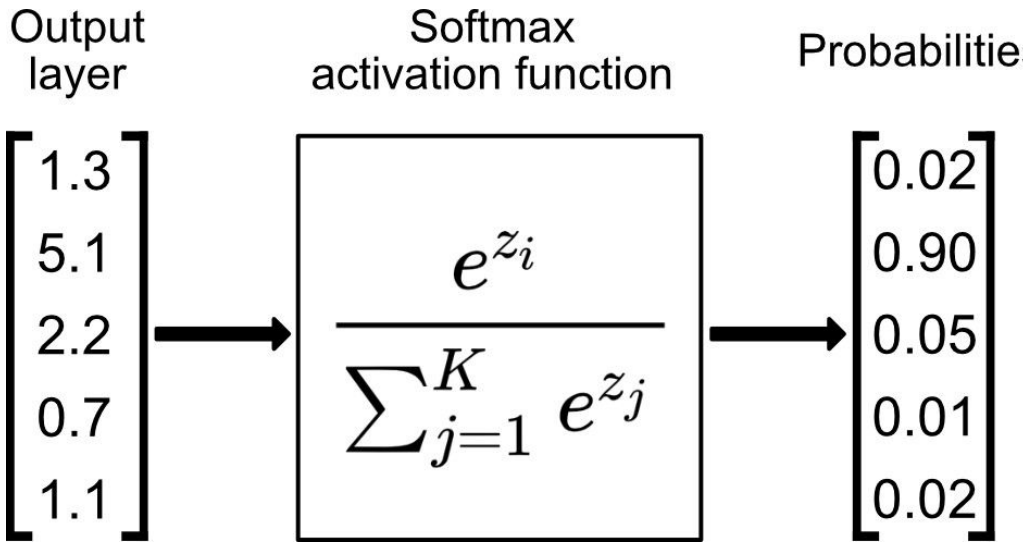
continuously differentiable

$$f(u) = \frac{1}{1 + e^{-\sigma u}} \qquad f : \Re \rightarrow [0,1]$$



Softmax function

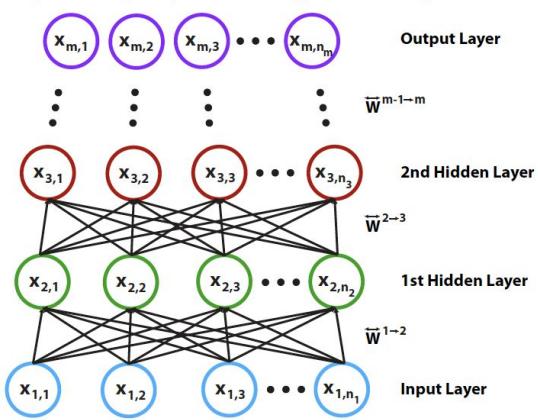
$$p\left(\vec{x}^{m}\right) = p\left(x_{m,1}, x_{m,2}, \cdots, x_{m,n_m}\right) = \left(\frac{e^{x_{m,1}}}{\sum_{j=1}^{n_m} e^{x_{m,j}}}, \frac{e^{x_{m,2}}}{\sum_{j=1}^{n_m} e^{x_{m,j}}}, \cdots, \frac{e^{x_{m,n_m}}}{\sum_{j=1}^{n_m} e^{x_{m,j}}} \right) = \frac{e^{\vec{x}^{m}}}{\sum_{j=1}^{n_m} e^{x_{m,j}}}$$



https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60

Feed Forward Neural Networks

- Information only flows forward, There are no feedback connections in which outputs of the model are fed back into itself
- Usually the number of neurons in consecutive layers decreases(lower level layers have more neurons than higher)
- For regression
  - The number of neurons at the final layers = 1
  - The single neuron at the output layer returns a continuous value
- For classification:
  - For classification into k classes, we set the number of units in the output layer to k
  - The neuron i in the output layer returns the probability that the input should be classified as class i.
  - Probabilities are computed using the softmax function



$$\vec{x}^{\,i} = \left(x_{i,1}, x_{i,2}, \cdots, x_{i,n_i}\right)^T$$

$$\vec{\vec{W}}^{i\to i+1} = \begin{bmatrix} w_{1,1}^{i\to i+1} & w_{1,2}^{i\to i+1} & \cdots & w_{1,n_i}^{i\to i+1} \\ w_{2,1}^{i\to i+1} & w_{2,2}^{i\to i+1} & \cdots & w_{2,n_i}^{i\to i+1} \\ \vdots & \vdots & \cdots & \vdots \\ w_{n_{i-1},1}^{i\to i+1} & w_{n_{i-1},2}^{i\to i+1} & \cdots & w_{n_{i-1},n_i}^{i\to i+1} \end{bmatrix}$$

Deep Learning Programming

Keras

- It doesn’t handle low-level computations itself; instead, it passes it to another library called the Backend. After version 2.4 only TensorFlow is supported.
- In the newer versions, Keras acts as an interface for the TensorFlow
- Supports distributed training of deep-learning models on clusters of GPUs and TPUs
- input_shape=(784,) is equal to batch_input_shape=(None, 784) is equal to input_dim=784
- input_shape=(10, 64) is equal to batch_input_shape=(None, 10, 64) is equal to input_dim=64

Sequential API

```
model = Sequential()
model.add(Dense(10,activation='relu')) # You don't need to specify input
model.add(Dense(5,activation='relu'))
model.add(Dense(1))
```

Functional API

- You define an input layer and specify the input size
- You define each layer by specifying a name, the number of neurons, activation function, etc.
- Put the previous layer in parentheses at the end of the expression

```
input_layer = Input(shape=(3,))

Layer_1 = Dense(4, activation="relu")(input_layer)

Layer_2 = Dense(4, activation="relu")(Layer_1)

output_layer= Dense(1, activation="linear")(Layer_2)

##Defining the model by specifying the input and output layers

model = Model(inputs=input_layer, outputs=output_layer)
```

Example: Keras

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

Example: CNN Model Using Keras

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32,
3)))
model.add(MaxPool2D())
model.add(Conv2D(16, (3, 3), activation='relu'))
model.add(MaxPool2D())
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

R Keras Package

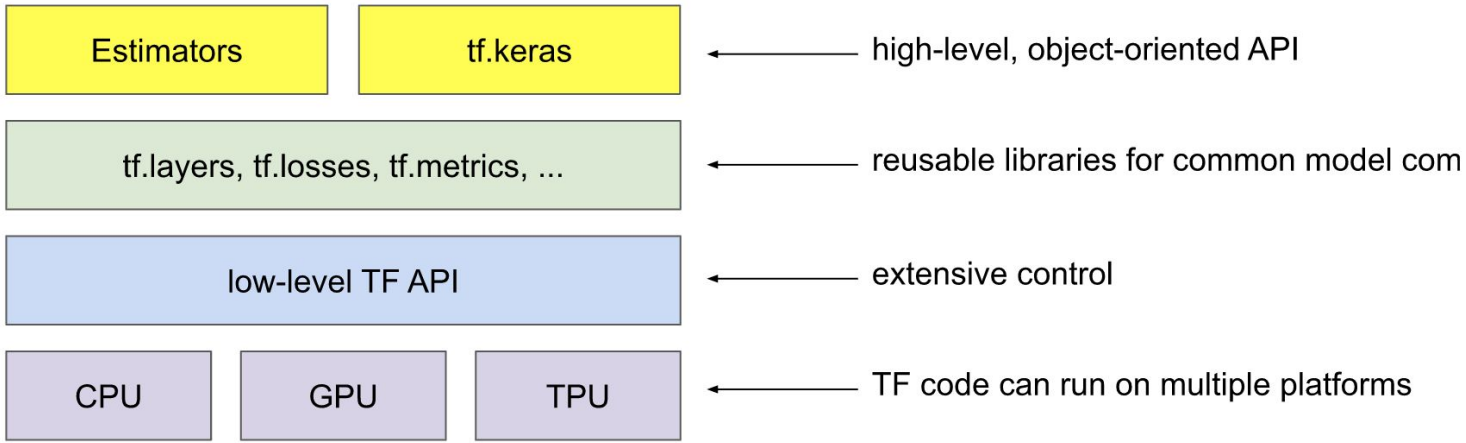
```
library(keras)

model <- keras_model_sequential()

model %>% layer_dense(units = 256, activation = 'relu',
input_shape = c(784)) %>%

        layer_dropout(rate = 0.4) %>%

        layer_dense(units = 128, activation = 'relu') %>%

        layer_dropout(rate = 0.3) %>%

        layer_dense(units = 10, activation = 'softmax')
```

```
 model %>% compile(
   loss = 'categorical_crossentropy',
   optimizer = optimizer_rmsprop(),
   metrics = c('accuracy')
 )

history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
 )
```

Example code:
https://cran.r-project.org/web/packages/keras/vignettes/index.html

Tensorflow High-Level APIs

- Tensorflow 2.0 has two high-level deep learning APIs:
  - tf.keras
  - Tf.estimator (not recommended for new code, use keras instead)



Source: https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit

TensorFlow 2

- TensorFlow 2.0 introduced many changes
  - Changed the automatic differentiation scheme (automatically calculating the gradient vector of a model with respect to each of its parameters) from the static computational graph to the Define-by-Run
  - TensorFlow includes an “eager execution” mode (operations are evaluated immediately as opposed to being added to a computational graph which is executed later).
  - Code executed eagerly can be examined step-by step-through a debugger, since data is augmented at each line of code rather than later in a computational graph
  - The session and placeholder objects do not exist anymore

Eager Execution

- In Tensorflow 1.0 the operations were performed inside a session
- Tensorflow 2.0 supports eager execution (no need for placeholders)

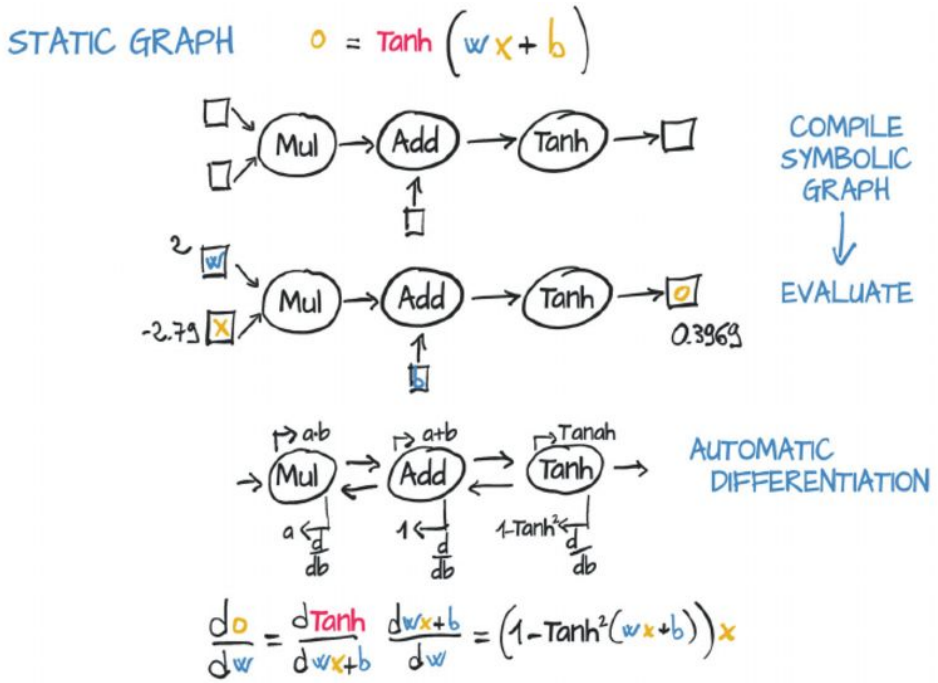
```
import tensorflow as tf

var1 = tf.constant(8)
var2 = tf.constant(9)
result_tf_1 = var1 + var2
sess = tf.Session()
print("The result for Tensorflow 1.0 is:",sess.run(result_tf_1))
```

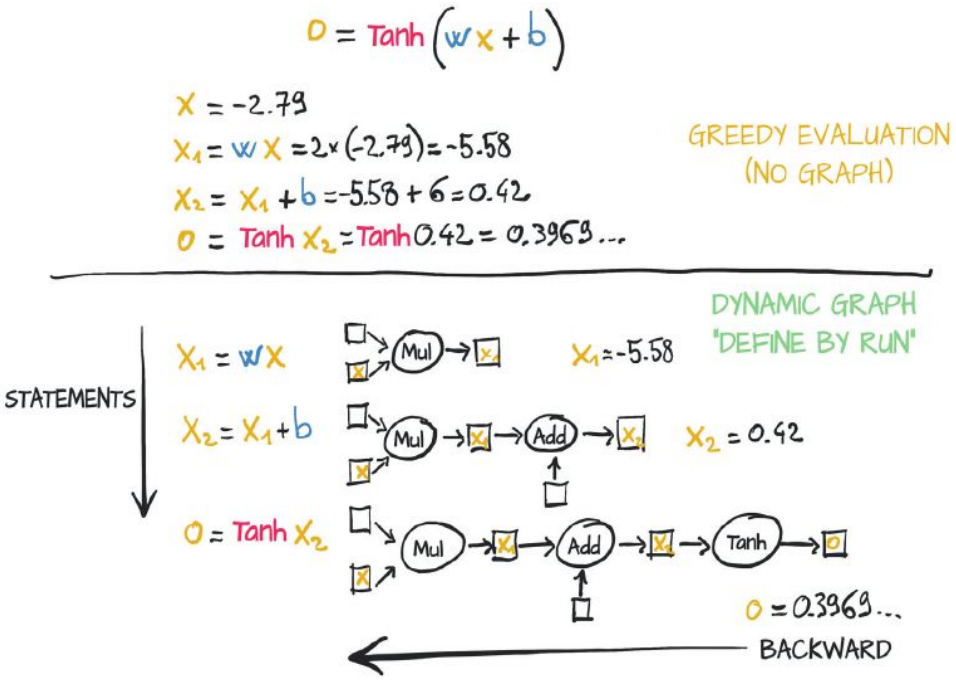
```
import tensorflow as tf

var3 = tf.constant(5)
var4 = tf.constant(6)
result_tf_2 = var3 + var4
print("The result of tensorflow 2 is:",result_tf_2)
```

Define-and-run vs static computational graph

- **Define-and-run** (Static computational graph): A model is constructed, before execution, as a computational graph structure, which later gets executed with different inputs while remaining fixed (Theano).
- **Define-by-run** (dynamic computational graph): general-purpose AD capability available where a model is a regular program in the host programming language, whose execution dynamically constructs a computational graph on-the-fly that can freely change in each iteration (PyTorch)

Define-and-run vs Define-by-run





Linear Regression: TensorFlow 2

```
import tensorflow as tf
import numpy as np

x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

W = tf.Variable(tf.random.normal([1]))
b = tf.Variable(tf.zeros([1]))

# mean squared error between y_data and computed y
def mse_loss():
    y = W * x_data + b
    loss = tf.reduce_mean(tf.square(y - y_data))
    return loss

optimizer = tf.keras.optimizers.Adam()
for step in range(5000):
    optimizer.minimize(mse_loss, var_list=[W,b])
    if step % 500 == 0:
        print(step, W.numpy(), b.numpy())
```

PyTorch

- Good debugging Support
- PyTorch provides two APIs:
  - Low-level Tensor computing (similar to NumPy)
  - High-level API
- PyTorch has a class called Tensor (torch.Tensor) to store and operate on homogeneous multidimensional rectangular arrays of numbers, similar to NumPy Arrays
  - PyTorch Tenors can be operated on a CUDA-capable NVIDIA GPU

Example: Low-level Tensor API

```
import torch

dtype = torch.float

device = torch.device("cpu") # This executes all calculations on the CPU

# device = torch.device("cuda:0") # This executes all calculations on the GPU


# Creation of a tensor and filling of a tensor with random numbers

a = torch.randn(2, 3, device=device, dtype=dtype)

b = torch.randn(2, 3, device=device, dtype=dtype)

print(a*b)
```

High Level API Example: Standard Neural Net

```
import torch

from torch import nn


class NeuralNetwork(nn.Module):

    def __init__(self): #  super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512), # Linear Layers have an input and output shape
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )


    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
      return logits
```

Neural networks are defined as classes, inherit from nn.Module class

Layers and variables are defined in the __init__ method

This function defines the forward propagation logic which implies the structure of the network

Example: CNN Using PyTorch

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3)
        self.conv2 = nn.Conv2d(32, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 10)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
       # view returns a new tensor with the same data as the self tensor but of a different shape.
        x = x.view(-1, 16 * 6 * 6)
        x = F.log_softmax(self.fc1(x), dim=-1)
        return x
```

Deep Learning Regularization

Weight Decay/Ridge Regression/Tikhonov regularization

- This regularization strategy drives the weights closer to the origin by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}),$$

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

- To take a single gradient step to update the weights, we perform this update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\right).$$

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}).$$

Keras/Pytorch Weight Decay

Keras Weight Decay (L2), L1 and L1_L2 Regularizers:

keras.regularizers.l1(0.01)

keras.regularizers.l2(0.01)

keras.regularizers.l1_l2(l1=0.01, l2=0.01)

model.add(Dense(32, kernel_regularizer=l2(0.01), bias_regularizer=l2(0.01)))

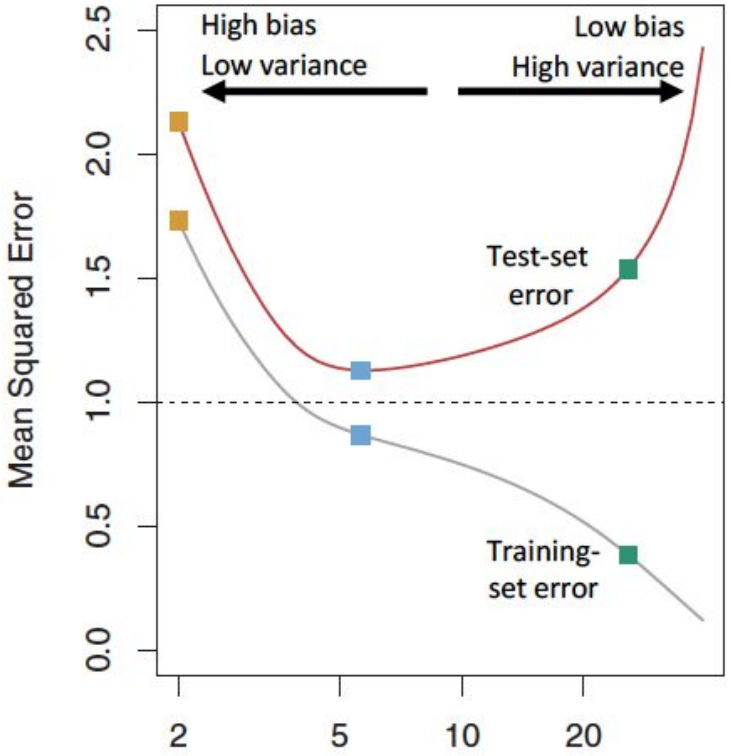
Pytorch Regularizers:

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3, weight_decay=1e-4)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)

Early Stopping

- Early stopping may be used either alone or in conjunction with other regularization strategies.
- Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed.
    - One strategy is to initialize the model again and retrain on all of the data. In this second training pass, we train for the same number of steps as the early stopping procedure determined was optimal in the first pass.



Early Stopping: Keras

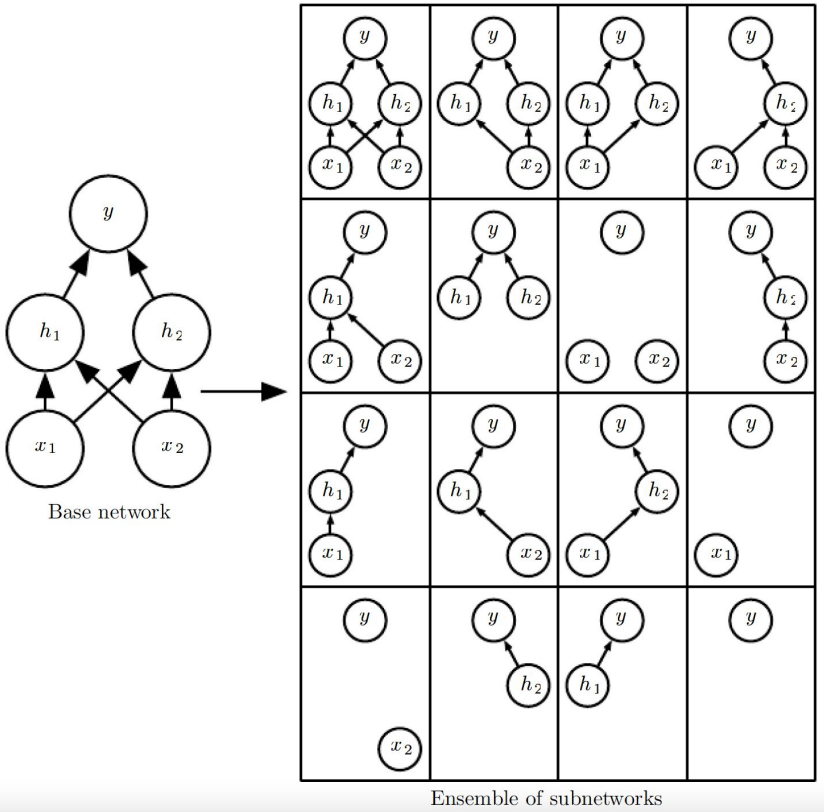
```
callback = [EarlyStopping(monitor='val_loss',patience=2),ModelCheckpoint(filepath='best_model.h5', monitor='val_loss', save_best_only=True)]

model = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])

model.compile(tf.keras.optimizers.SGD(), loss='mse')

history = model.fit(X,Y, epochs=10, batch_size=1, callbacks=[callback])
```

Early Stopping: Pytorch

```
def traindata(device, model, epochs, optimizer, loss_function, train_loader, valid_loader):

…
    # Early stopping
     current_loss = validation(model, device, valid_loader, loss_function)
     print('The Current Loss:', current_loss)

     if current_loss > last_loss:
        trigger_times += 1
        print('Trigger Times:', trigger_times)

        if trigger_times >= patience:
            print('Early stopping!\nStart to test process.')
            return model

     else:
        print('trigger times: 0')
        trigger_times = 0

     last_loss = current_loss
```

Dropout

- Dropout provides an inexpensive approximation to training and evaluating a **bagged** ensemble of exponentially many neural networks.
- Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network
- In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.



Dropout: Keras

```
model = Sequential()
model.add(Dense(60, input_shape=(60,), activation='relu',
kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(30, activation='relu',
kernel_constraint=MaxNorm(3)))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Dropout: PyTorch

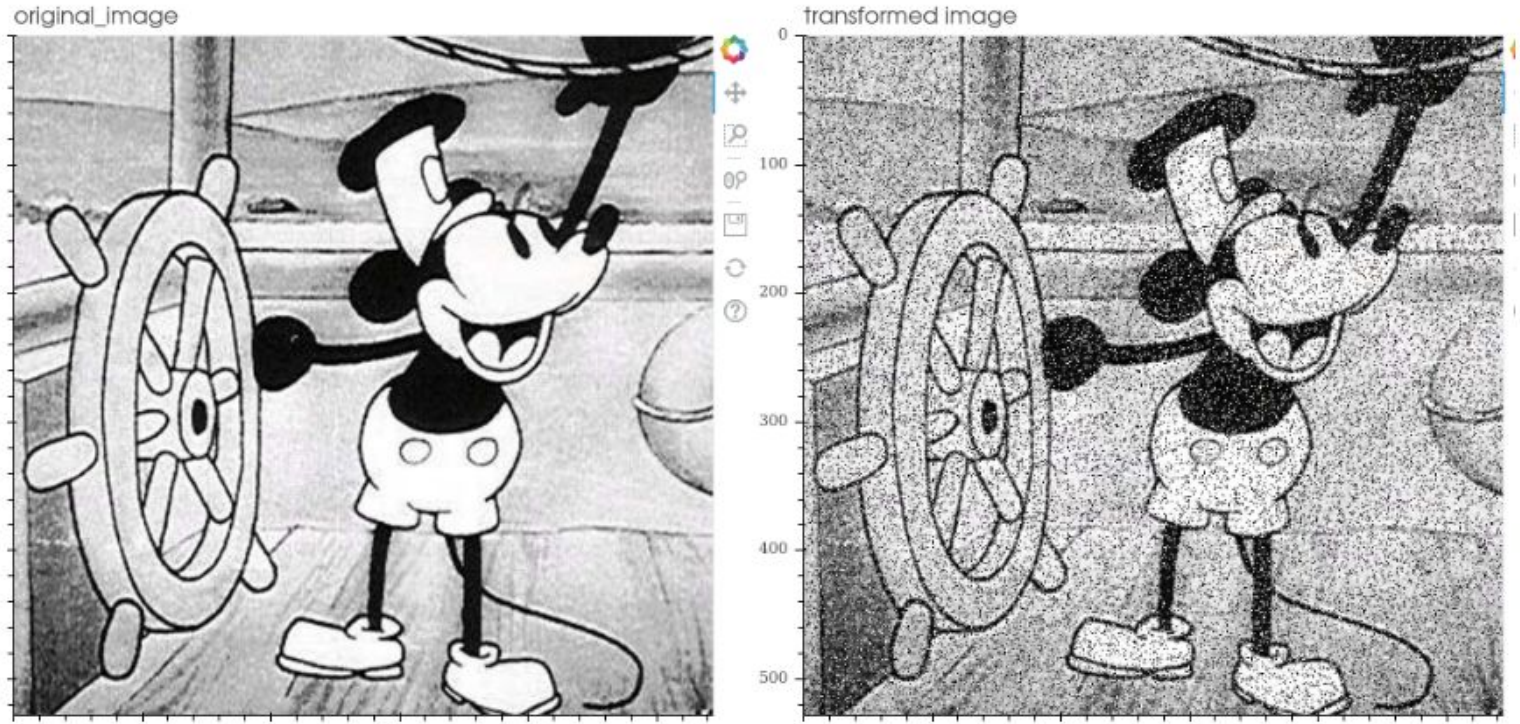
```
def __init__(self):
    super(Net1, self).__init__()
    self.hid1 = T.nn.Linear(4, 8)
    self.drop1 = T.nn.Dropout(0.50)
    self.hid2 = T.nn.Linear(8, 8)
    self.drop2 = T.nn.Dropout(0.25)
    self.oupt = T.nn.Linear(8, 1)
```

Dataset Augmentation: Image

- One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between ‘b’ and ‘d’ and the difference between ‘6’ and ‘9’, so horizontal flips and 180◦ rotations are not appropriate ways of augmenting datasets for these tasks.

https://www.kaggle.com/getting-started/190280

Dataset Augmentation: Noise Injection Example



https://github.com/nvblanco/ida_lib

Data Augmentation in Keras

```
data_augmentation = tf.keras.Sequential([
layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
layers.experimental.preprocessing.RandomRotation(0.2)])
```

```
datagen = ImageDataGenerator(rotation_range=90)
```

Data augmentation in PyTorch

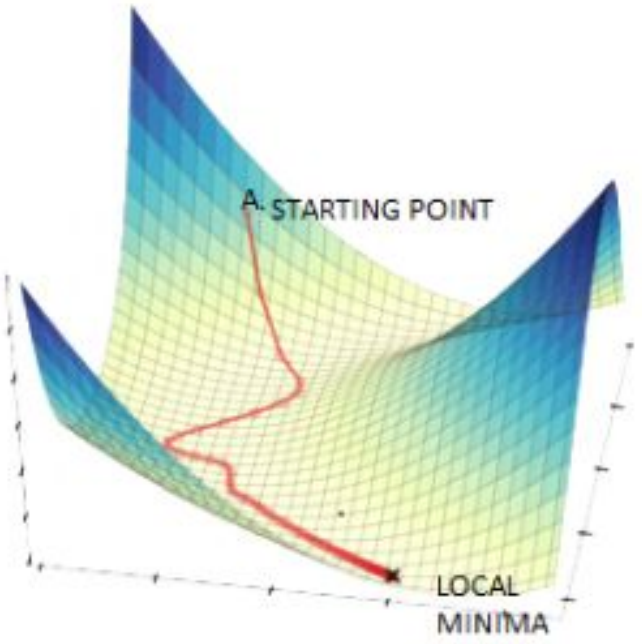
- Transforms library from the torchvision package has popular datasets, model architectures, and common image transformations for Computer Vision tasks

```
from torchvision import transforms as tr

from torchvision.transfroms import Compose

pipeline = Compose(

             [tr.RandomRotation(degrees = 90),

              tr.RandomRotation(degrees = 270)])

 augmented_image = pipeline(img = img)
```

Deep Learning Optimization

Optimization Goal

- We want to converge faster and to a better minima
- Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function J(θ) in the hope that doing so will improve P. This is in contrast to pure optimization, where minimizing J is a goal in and of itself.
  - If we knew the true distribution $p_{data}(x, y)$, risk minimization would be an optimization task solvable by an optimization algorithm.
  - However, when we do not know $p_{data}(x, y)$ but only have a training set of samples, we have a machine learning problem.



Optimizers Cheat Sheet

Gradient Descent:	$\theta = \theta - \alpha \, . \nabla_\theta J(\theta)$
Stochastic Gradient Descent	$\theta = \theta - \alpha \, . \nabla_\theta J(\theta; sample)$
Mini-Batch Gradient Descent	$\theta = \theta - \alpha \, . \nabla_\theta J(\theta; N \, samples)$
SGD + Momentum	$v = \gamma . v + \eta . \nabla_\theta J(\theta)$ $\theta = \theta - \alpha \, v$
SGD + Momentum + Acceleration	$v = \gamma . v + \eta . \nabla_\theta J(\theta - \gamma . v)$ $\theta = \theta - \alpha \, v$
Adagrad	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon} \nabla_{\theta_{t,i}} J(\theta_{t,i})$
Adadelta	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \nabla_{\theta_{t,i}} J(\theta_{t,i})$
Adam	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}] + \epsilon}} \times \mathrm{E}[g_{t,i}]$

Keras Example

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
```

```
keras.optimizers.Adagrad(lr=0.01, epsilon=None, decay=0.0)
```

```
keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)
```

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Pytorch Example

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

optimizer = optim.Adam(model.parameters(), lr=0.0001)

optimizer= optim.RMSprop(model.parameters(), lr=0.0005)
```

Batch normalization

- Batch normalization is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling (Let H be a minibatch of activations of the layer to normalize):

$$H' = \frac{H - \mu}{\sigma},$$

$$\mu = \frac{1}{m}\sum_i H_{i,:}$$

$$\sigma = \sqrt{\delta + \frac{1}{m}\sum_i (H - \mu)_i^2},$$

Where δ is a small positive value such as 10^{-8}, imposed to avoid encountering the undefined gradient of √z at z= 0.

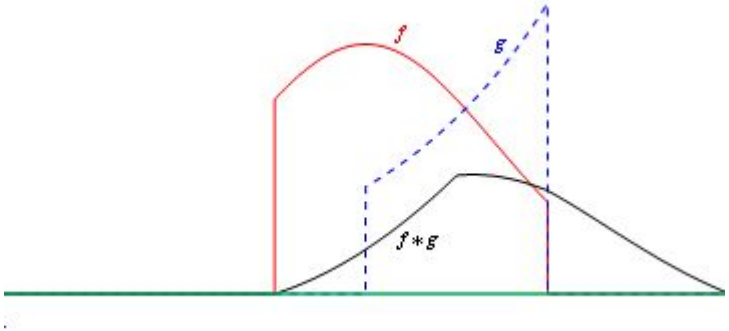
Keras Example

```
model = Sequential()

model.add(Dense(10, activation="relu", input_shape=(X_train.shape[1],)))

model.add(BatchNormalization())

model.add(Dense(8, activation="relu"))

model.add(BatchNormalization())

model.add(Dense(1, activation="sigmoid"))
```

PyTorch Example

```
class MLP(nn.Module):
  def __init__(self):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Flatten(),
      nn.Linear(32 * 32 * 3, 64),
      nn.BatchNorm1d(64),
      nn.ReLU(),
      nn.Linear(64, 32),
      nn.BatchNorm1d(32),
      nn.ReLU(),
      nn.Linear(32, 10)
     )
```

Convolutional Neural Networks



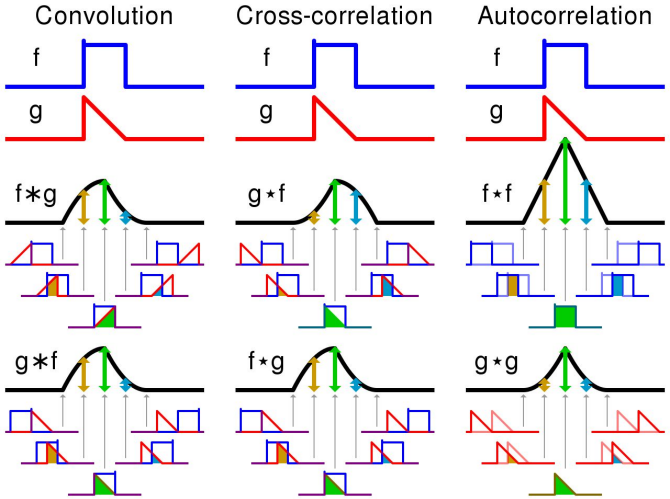
$$s(t) = \int x(a)w(t-a)da.$$

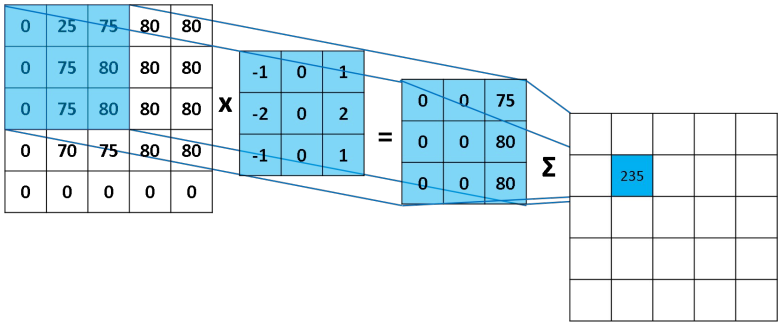
$$s(t) = (x * w)(t).$$

$$S(i,j) = (I * K)(i,j) = \sum\sum I(m,n)K(i-m, j-n).$$

- many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel

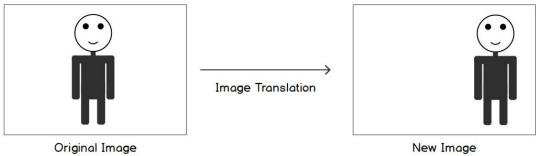
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n).$$

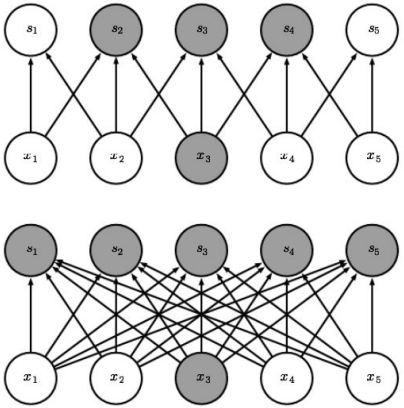




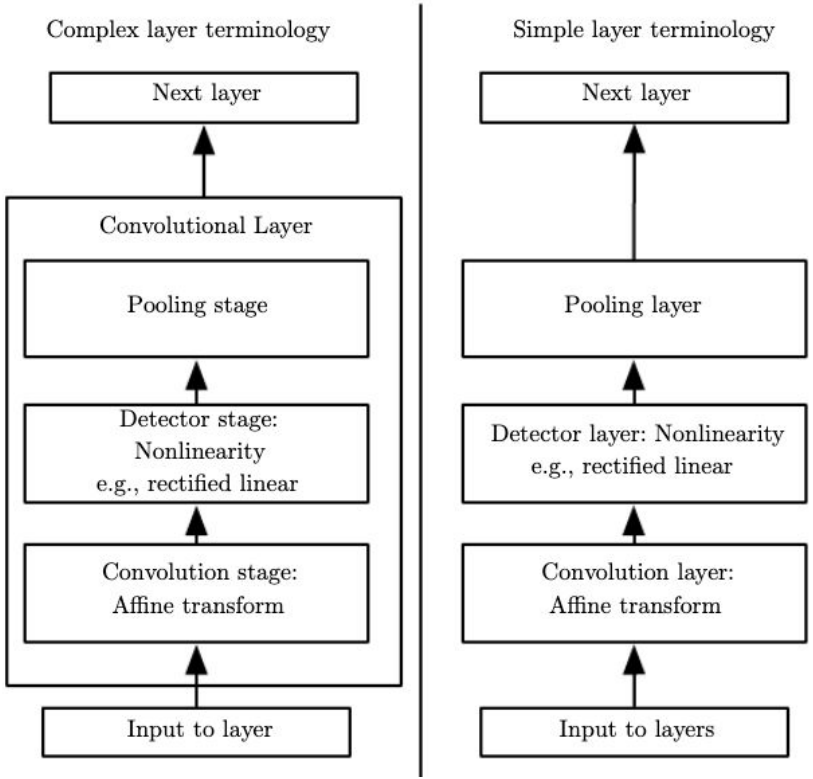
CNN

- Specialized kind of neural network for processing data that has a known grid-like topology, Examples include
  - Time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals
  - Image data,which can be thought of as a 2-D grid of pixels.
- Convolution leverages three important ideas that can help improve a machine learning system
  - Sparse interactions
  - Parameter sharing
    - Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited.
  - Equivariant representations



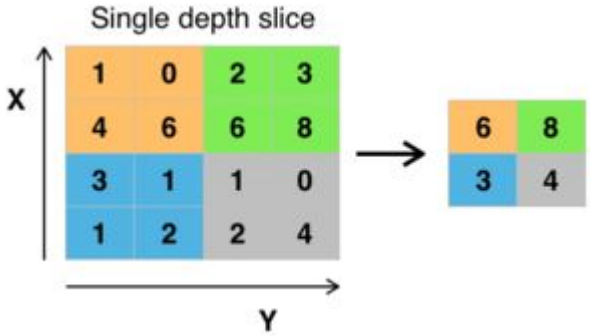


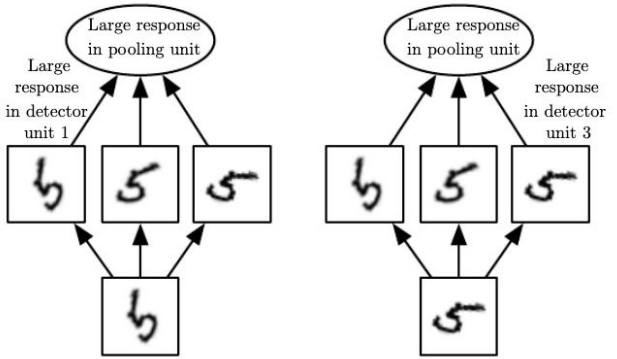
CNN Blocks


Complex layer terminology
Next layer
Convolutional Layer
Pooling stage
Detector stage:
Nonlinearity
e.g., rectified linear
Convolution stage:
Affine transform
Input to layer
Simple layer terminology
Next layer
Pooling layer
Detector layer: Nonlinearity
e.g., rectified linear
Convolution layer:
Affine transform
Input to layers

pooling

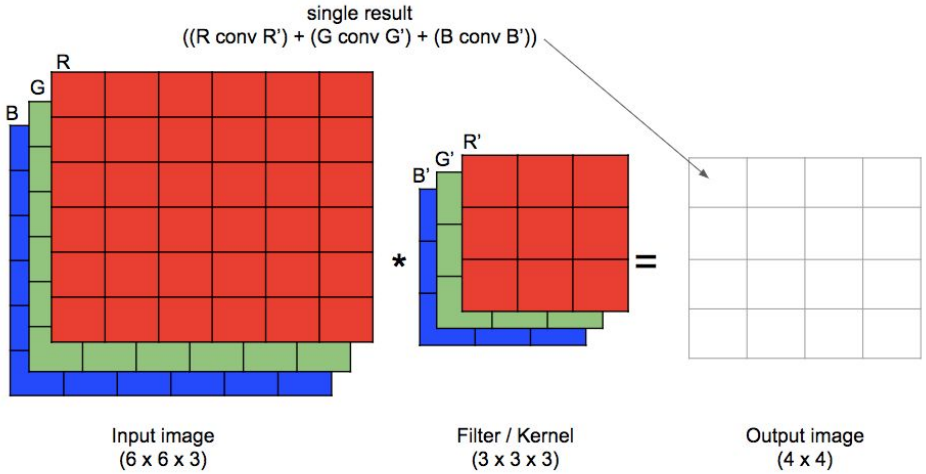
- pooling helps to make the representation approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.





From One Convolutional Layer to The Next

- Assume, we have a color image (RGB) of size 224*224 as input.
- The input to the first conv will be 224*224*3
- Assume we have 32 kernels of size 3*3 at the first conv layer. if we have “same” padding, then the output of this layer will be of dimensions 224*224*32.
- Note that our kernels will in fact be 3*3*3 because of the three channels in the input
    - In fact the 3 channels of the input (RGB) are being merged/combined into one matrix (3d dot operator)

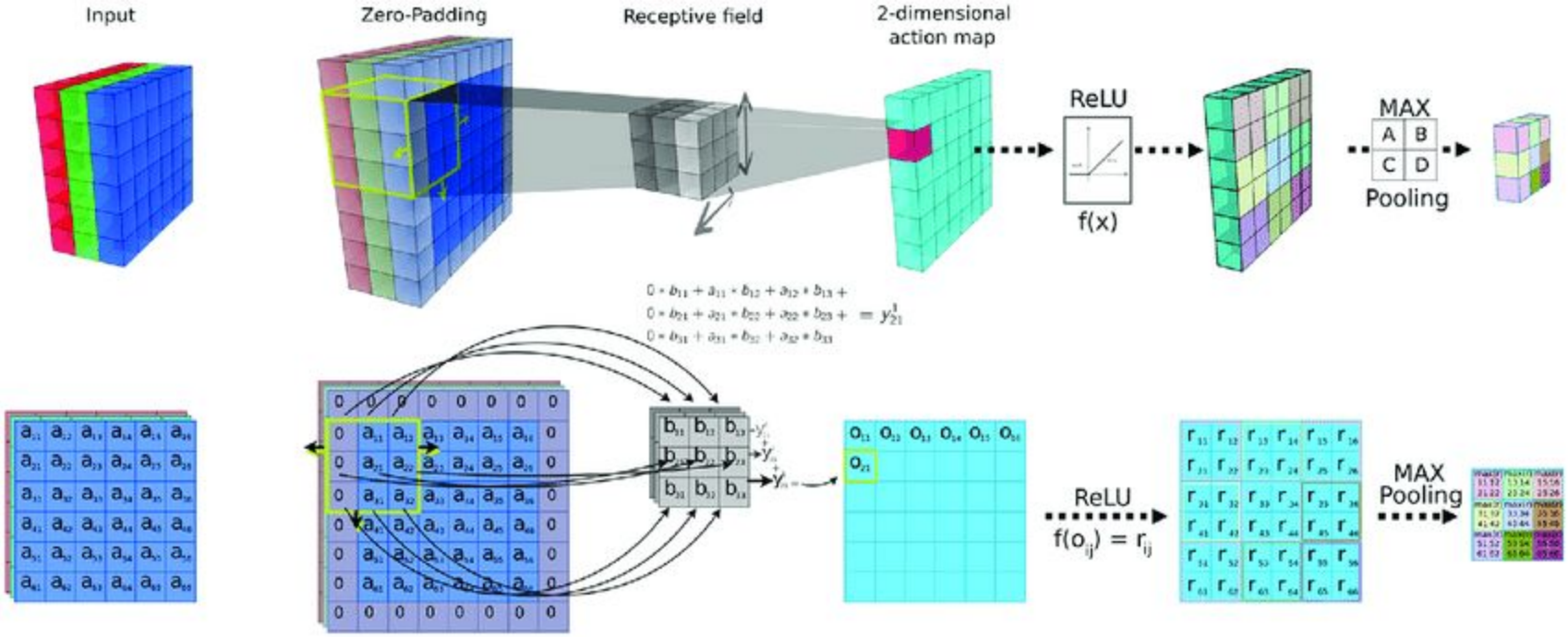


https://blog.devgenius.io/deep-learning-in-gradient-descent-style-part-2-e159e2cf8a99

From One Convolutional Layer to The Next

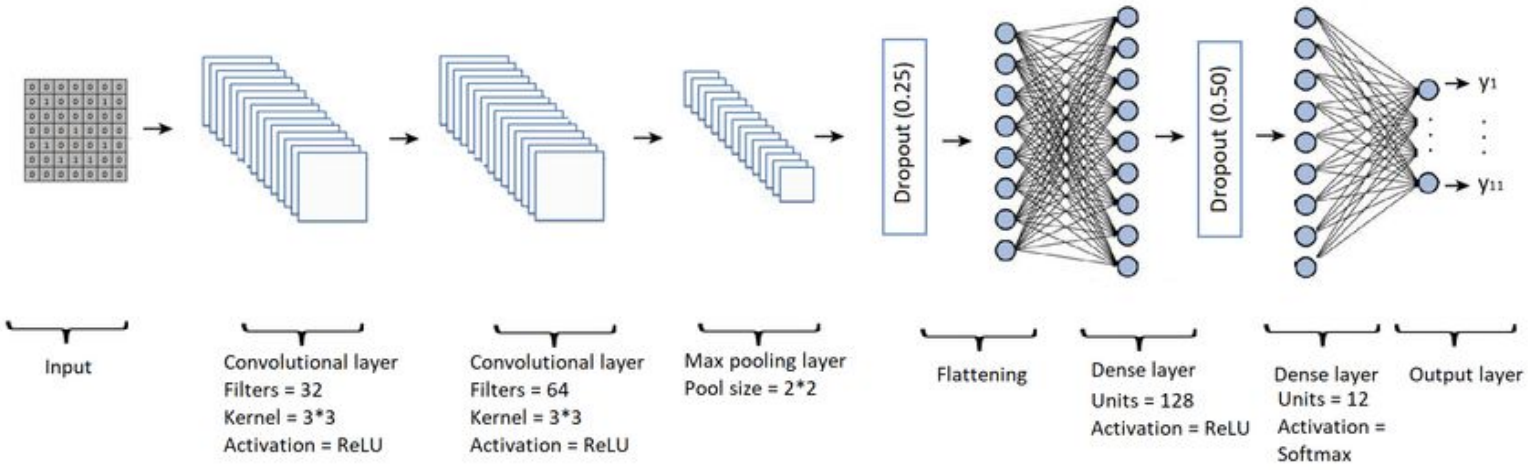
- Now assume that in the second convolutional layer, we have 64 kernels of 3*3 with a “same” padding. Thus we will have 64 outputs of size 224*224 or the output will be of size 224*224*64. Note that our kernels are in fact of size 3*3*32, because the output of the previous layer has 32 channels.
- The number of kernels of the previous layers become the number of channels for the next layer.
    - If we would generate a separate output for each channel we would have 32 outputs for the first layer and 32*64 for the second layer

CNN Architectures



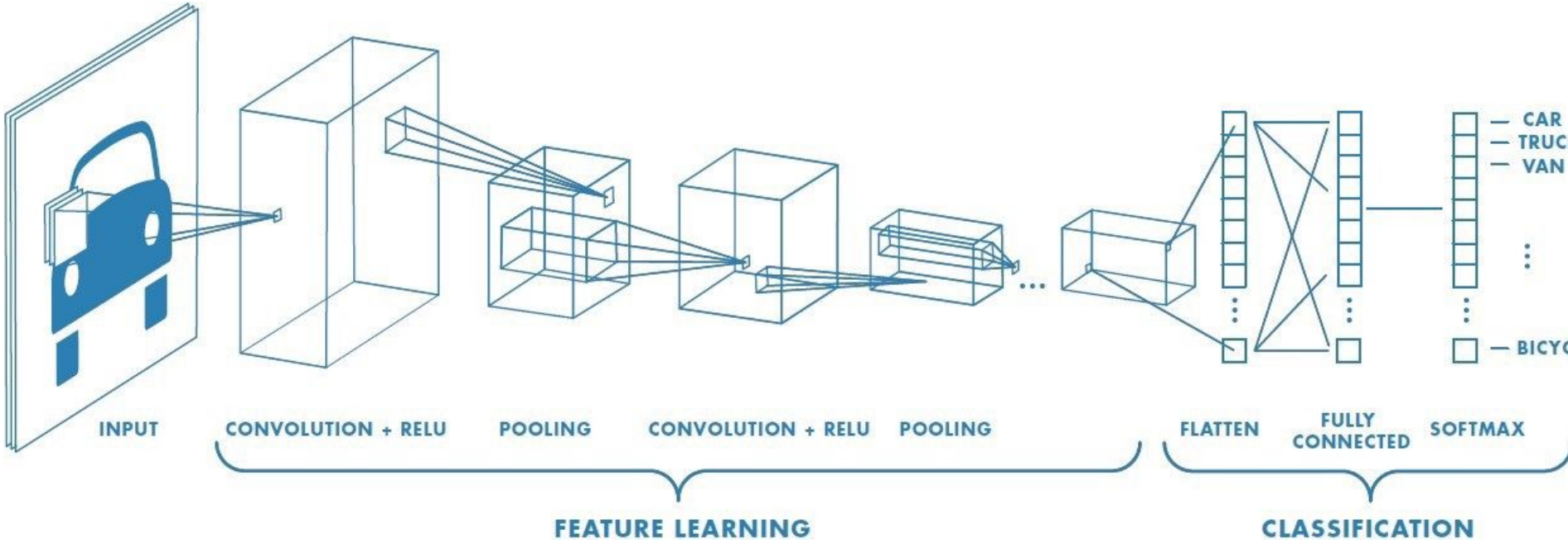
https://www.researchgate.net/figure/Operation-in-a-convolutional-layer-with-ReLU-and-Pooling-The-first-column-shows-a-6_fig1_322848501

CNN Architectures



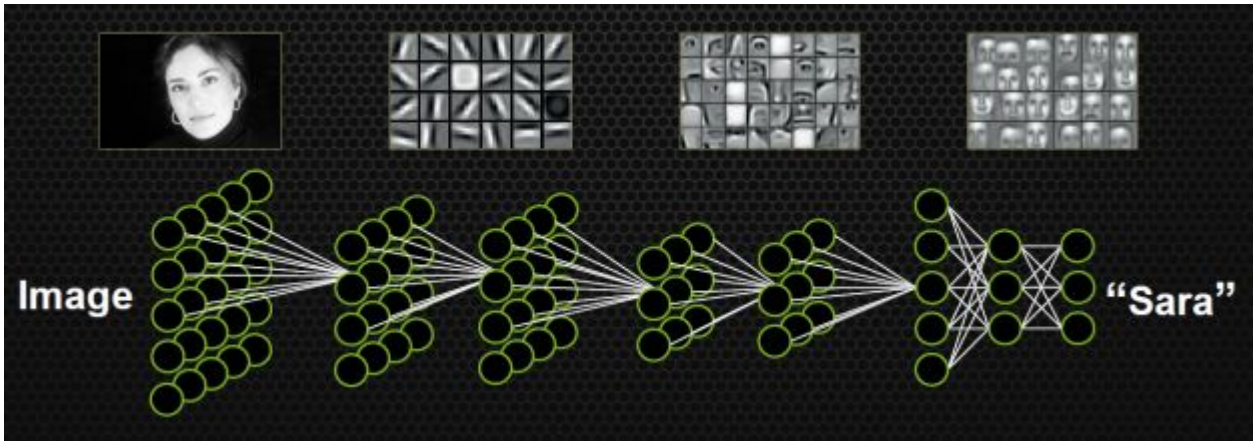
CNN Parameters

- Kernel size
- Number of kernels
  - Earlier layers (near the input) tend to have fewer filters while higher layers have more. This is done because the feature map size decreases with depth To equalize computation at each layer, the product of feature values with pixel position is kept roughly constant across layers. :
    32* (128*128)=64* (90*90)
  - Activation Number: number of feature maps times number of pixel positions
- Padding
  - For example, for the convolutional kernels of size 3x3 would receive a 2-pixel pad (1 pixel on each side of the image)
  - Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer.
  - Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network.
- Stride
- Pooling type: max, average
- Pooling size:
  - Max pooling downsamples image greatly. We usually use a 2×2 max-pooling For larger images, we can use 4×4 pooling in the lower layers
- Dilation
  - It tries to reduce the processing/memory without significant signal loss.
  - A dilation of 2 on a 3x3 kernel expands the kernel to a 5x5 window, while still processing 9 pixels (evenly spaced).


CAR
TRUC
VAN
BICYC
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION

Visualizing CNN Feature Maps

- To visualize feature maps, we apply the learned kernel filters to a sample input image



https://developer.nvidia.com/blog/accelerate-machine-learning-cudnn-deep-neural-network-library/

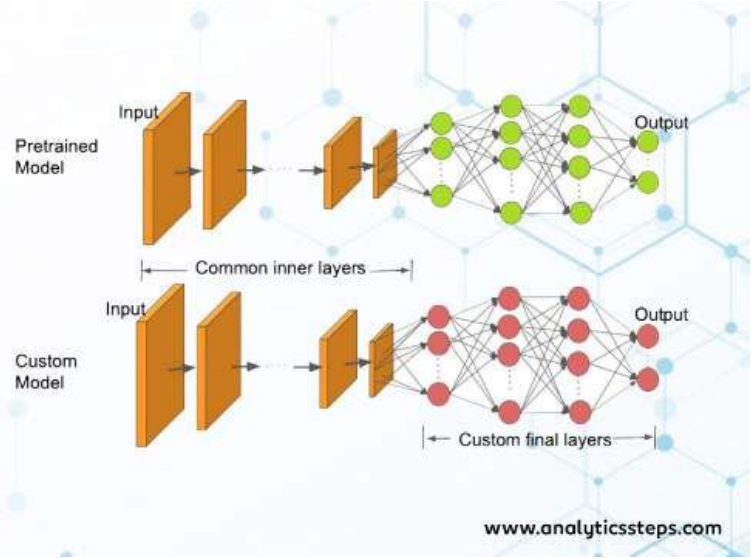
Demo: CNNs on MNIST Fashion

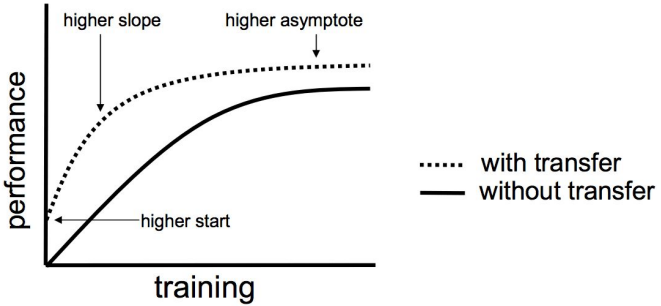
http://localhost:8888/notebooks/jupyter-notebooks/chapman-generative-AI/NN-review-CNN-MNIST-Fashion.ipynb

Pretrained Models and Transfer Learning

Transfer Learning

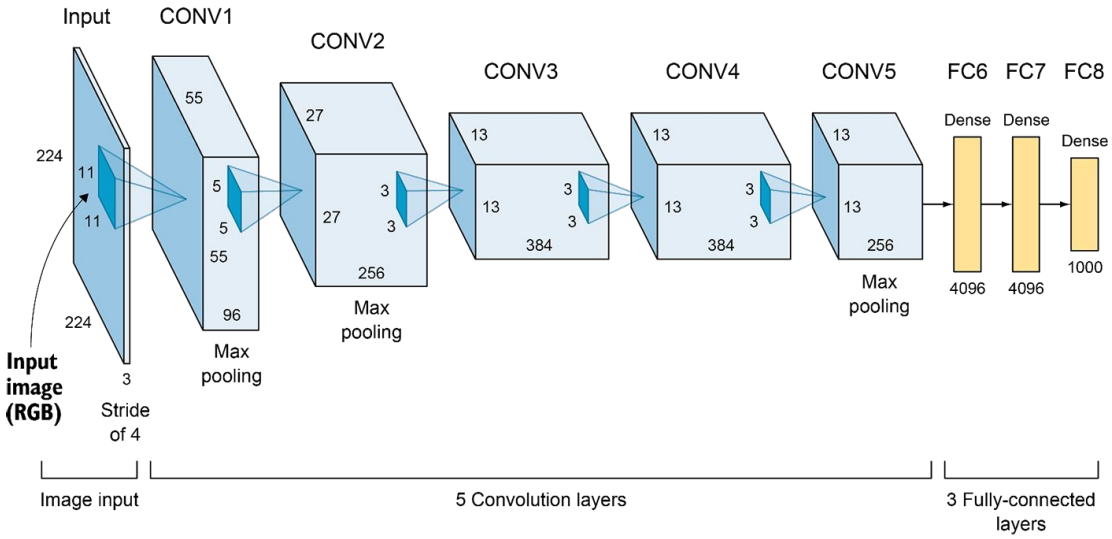
- We can also add a few new layers to the end of the existing model Using the data available for the new task, we tune the existing weights of the model to adapt for the new task and/or train the new added layers
- We might want to freeze some or all layers borrowed from the existing pre-trained model to prevent those weights from being updated
- Fine-tuning of the Whole Model





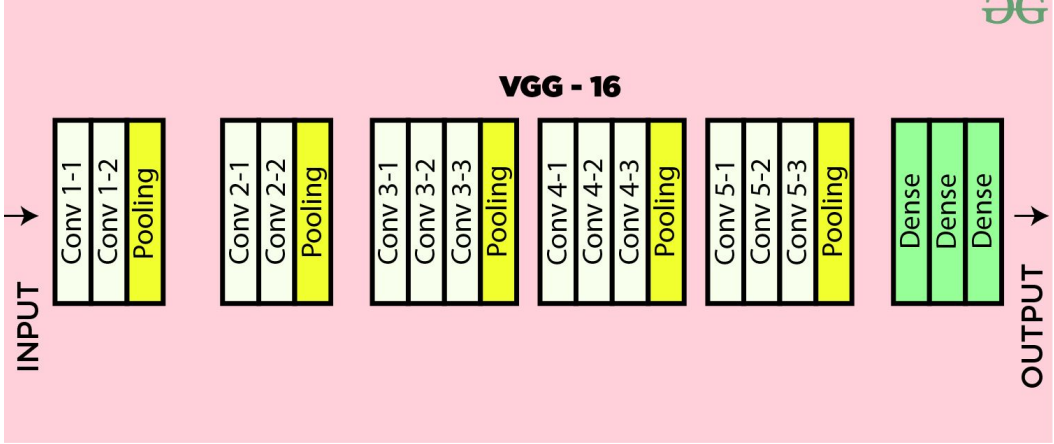
AlexNet

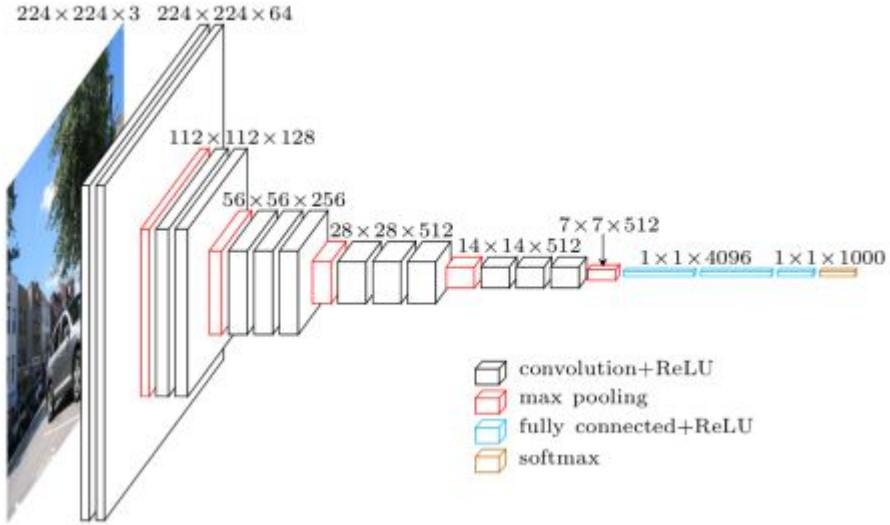
- It achieved a top-5 error of 15.3% in ImageNet
- AlexNet architecture consists of 5 convolutional layers, 3 max-pooling layers, 2 normalization layers, 2 fully connected layers, and 1 softmax layer.
- ReLU activation function is used in each convolutional layer.
- The input size is 224x224x3
- Has around 60 million parameters.



https://livebook.manning.com/book/deep-learning-for-vision-systems/chapter-5/v-3/35

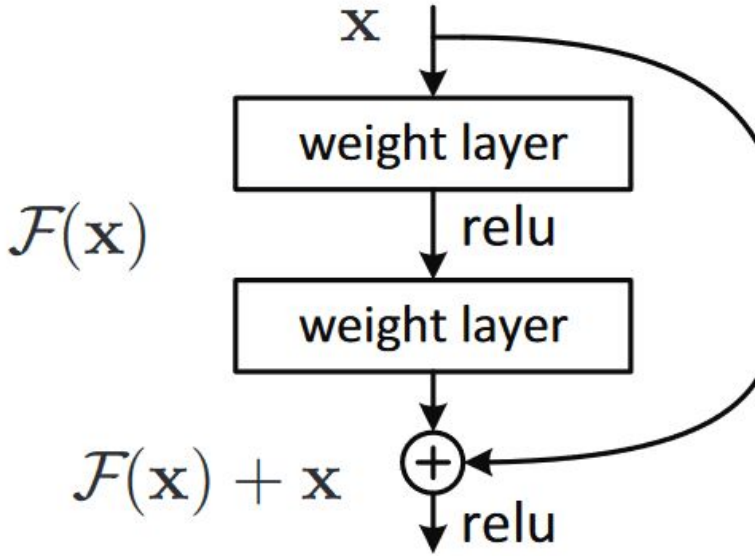
VGG16

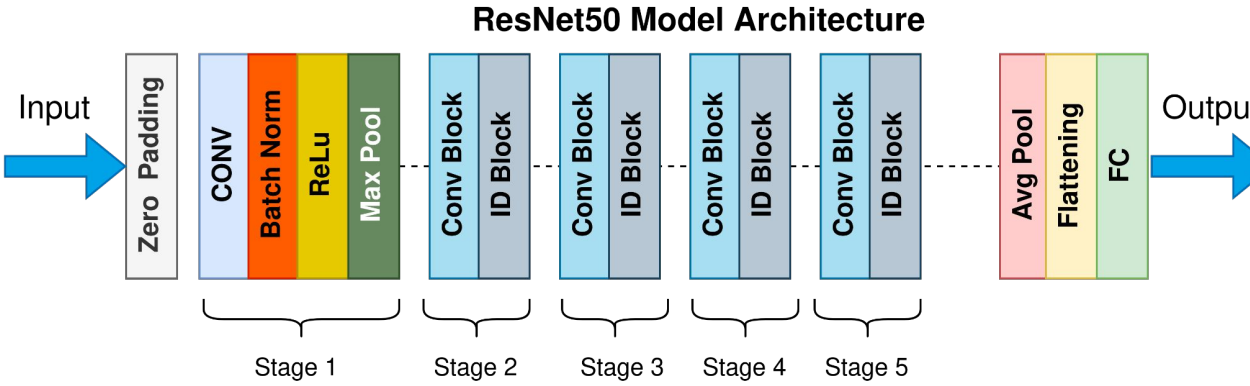

VGG - 16
INPUT
Conv 1-1
Conv 1-2
Pooling
Conv 2-1
Conv 2-2
Pooling
Conv 3-1
Conv 3-2
Conv 3-3
Pooling
Conv 4-1
Conv 4-2
Conv 4-3
Pooling
Conv 5-1
Conv 5-2
Conv 5-3
Pooling
Dense
Dense
Dense
OUTPUT


224×224×3
224×224×64
112×112×128
56×56×256
28×28×512
14×14×512
7×7×512
1×1×4096
1×1×1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

Skip Connection

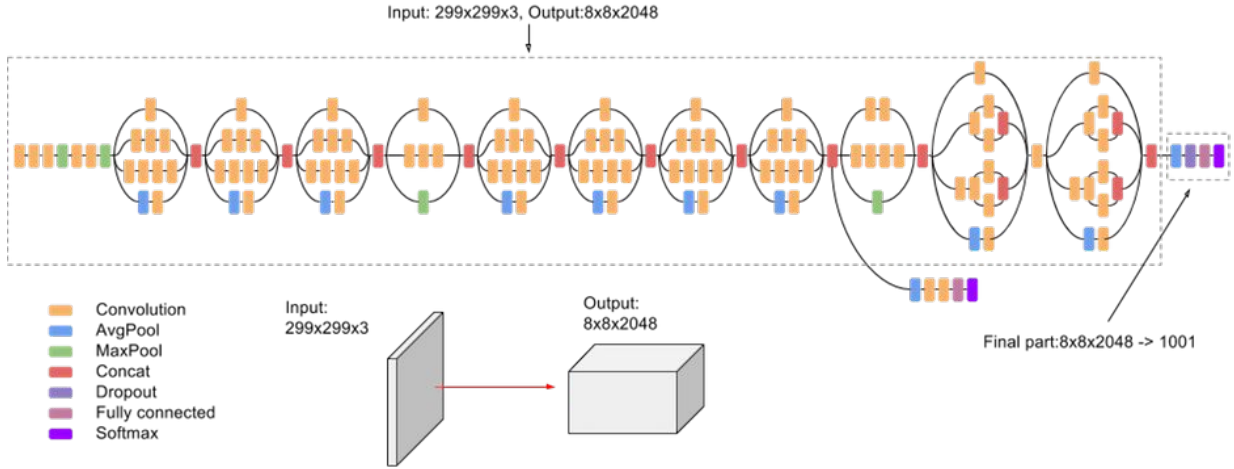
- layers feed into the next layer and also directly into the layers 1, 2, or 3 or more hops away.
- These are called residual blocks
- ResNets are build using these residual blocks





Inception V3

- The main idea behind the Inception model is to calculate multiple different conversions parallelly and concatenate them as a single output to be used by the next layer.
  - For instance, convolutions of sizes 1×1, 3×3, 5×5 and also a 3×3 max-pooling are done in parallel on the input, and the outputs from all of these operations are stacked together to generate the output of the model.



Demo: Transfer Learning in Computer Vision

http://localhost:8888/notebooks/jupyter-notebooks/chapman-generative-AI/NN-review-Transfer-Learning.ipynb

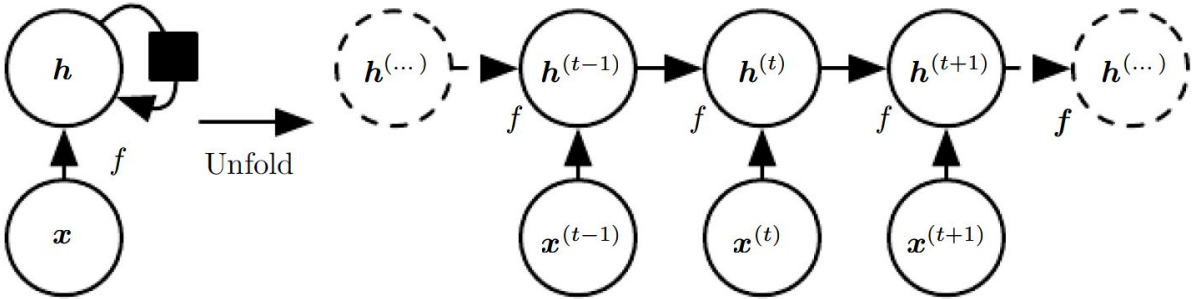
RNN

RNN

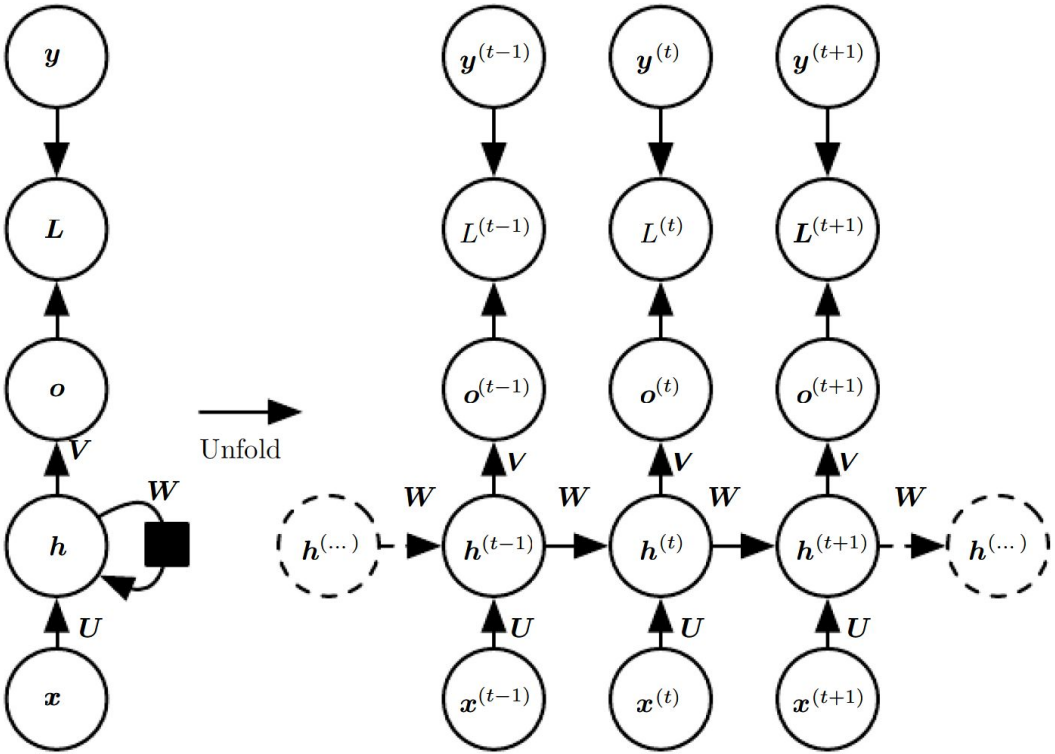
- Specialized for processing a sequence of values $x_1, \ldots, x_n$ such as words in a text
- RNNs have "memory" (store information from the previous data points in the input)
- Parameter sharing makes it possible to extend and apply the model to samples of different lengths
  - An RNN shares the same weights across several time steps
- Applications
  - Text autofill
  - Music generation
  - Speech recognition
  - Text analysis
  - Text summarization
  - RNNs can be combined with CNNs for tasks such as image to text

Unfolding the Graph

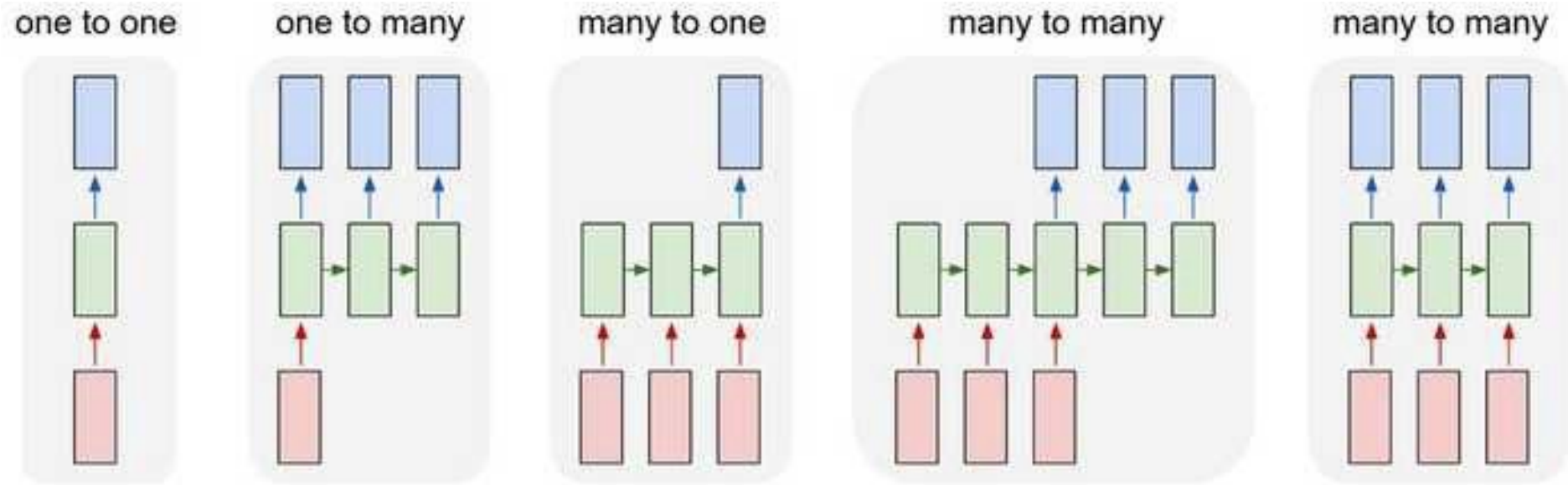
- Two ways to show a Recurrent relationship



- The black square indicates a delay of a single time step.


y
L
o
V
W
h
U
x
Unfold
$y^{(t-1)}$
$y^{(t)}$
$y^{(t+1)}$
$L^{(t-1)}$
$L^{(t)}$
$L^{(t+1)}$
$o^{(t-1)}$
$o^{(t)}$
$o^{(t+1)}$
$h^{(\cdots)}$
$h^{(t-1)}$
$h^{(t)}$
$h^{(t+1)}$
$h^{(\cdots)}$
$x^{(t-1)}$
$x^{(t)}$
$x^{(t+1)}$

RNN Types Based on the Input and Output



Traditional neural networks

music generation

sentiment analysis

machine translation

LSTM

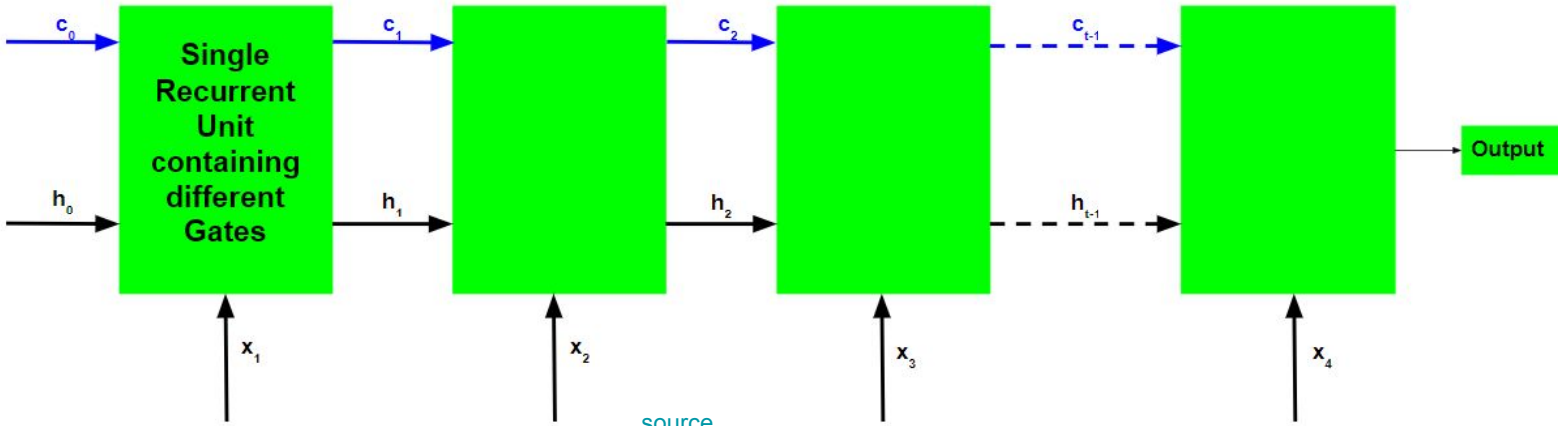
The Vanishing/Exploding Gradient Problem

- In a deep network, the gradients are propagated back in time all the way to the initial layer
- Since, we apply the chain rule in gradient calculation, gradients coming from deeper layers go through multiple matrix multiplications.
  - Thus, if they have small values (gradient<1) they diminish exponentially till they vanish (vanishing gradient)
  - On the other hand, if they have large values (gradient>1), then they grow exponentially large (exploding gradient)
- The long-term gradients in RNN which are back-propagated can "vanish" because of the computations involved in the process, which use finite-precision numbers
- Exploding/Vanishing gradient causes troubles in model training

Long Short Term Memory (LSTM)

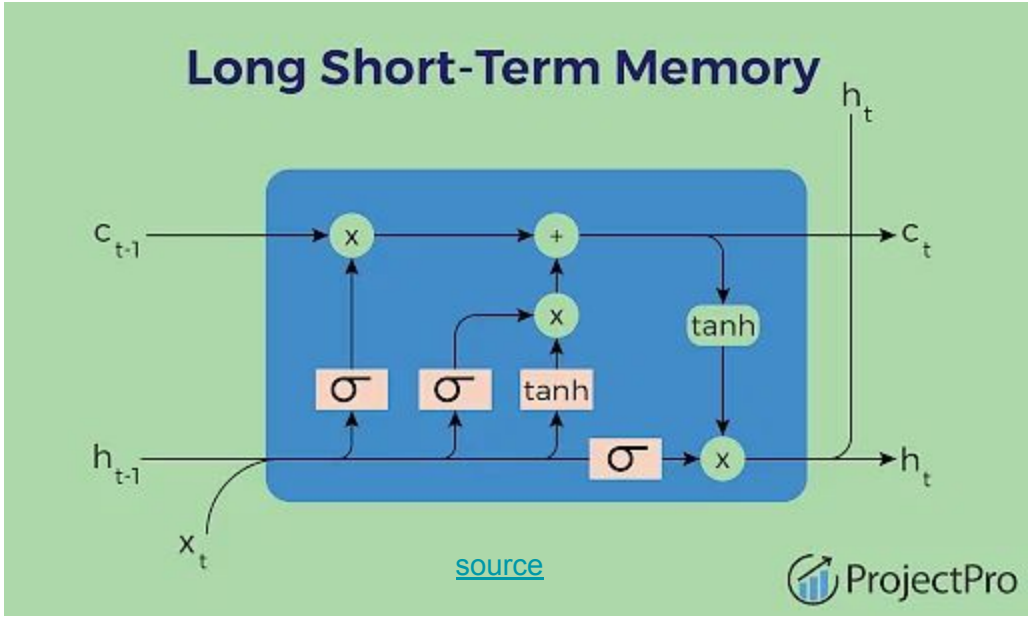
- Addresses the vanishing gradient problem in RNNs
- Use three types of gates: input, output, and forget
  - These gates decide which information to retain similar to GRU
- The recurrent units in an LSTM tries to **remember** all the past knowledge that the network has seen so far and to **forget** irrelevant data.
- LSTMs are the most popular variation of RNNs.
- Each LSTM recurrent unit has an additional vector called *Internal Cell State* which holds the information chosen to be remembered by the previous LSTM recurrent unit.
  - The internal cell state is also passed forward in addition to the hidden state

- Each unit has three inputs previous cell state (c_{t-1}), previous hidden state (h_{t-1}), and the current input (x_t)
- In addition to the unit's possible output (o_t), each unit has two outputs c_t and h_t



source

One LSTM Unit



Demo: Spam Detection Using LSTM

http://localhost:8888/notebooks/jupyter-notebooks/chapman-generative-AI/NN-review-RNN-LSTM.ipynb