Training Deep Networks

Lecture 11

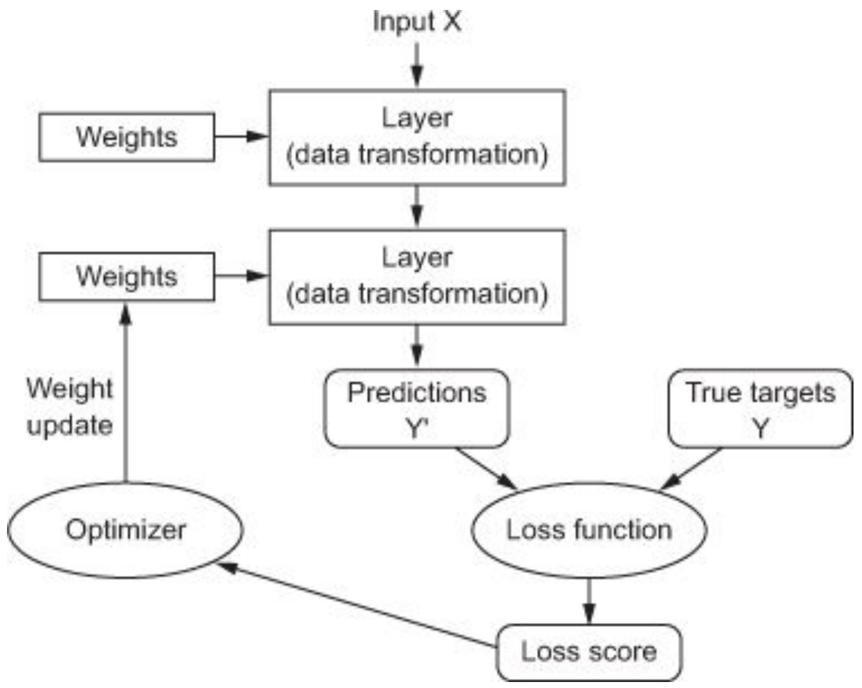
Last time: Learning Objectives

- ✓ Learn about the story behind the Artificial Neural Network
- ✓ Know the Linear Threshold Unit and Perceptron
- ✓ Expand to the Multilayer Perceptron (MLP)
- ✓ Understand how backpropagation works
- ✓ Finetune the neural networks

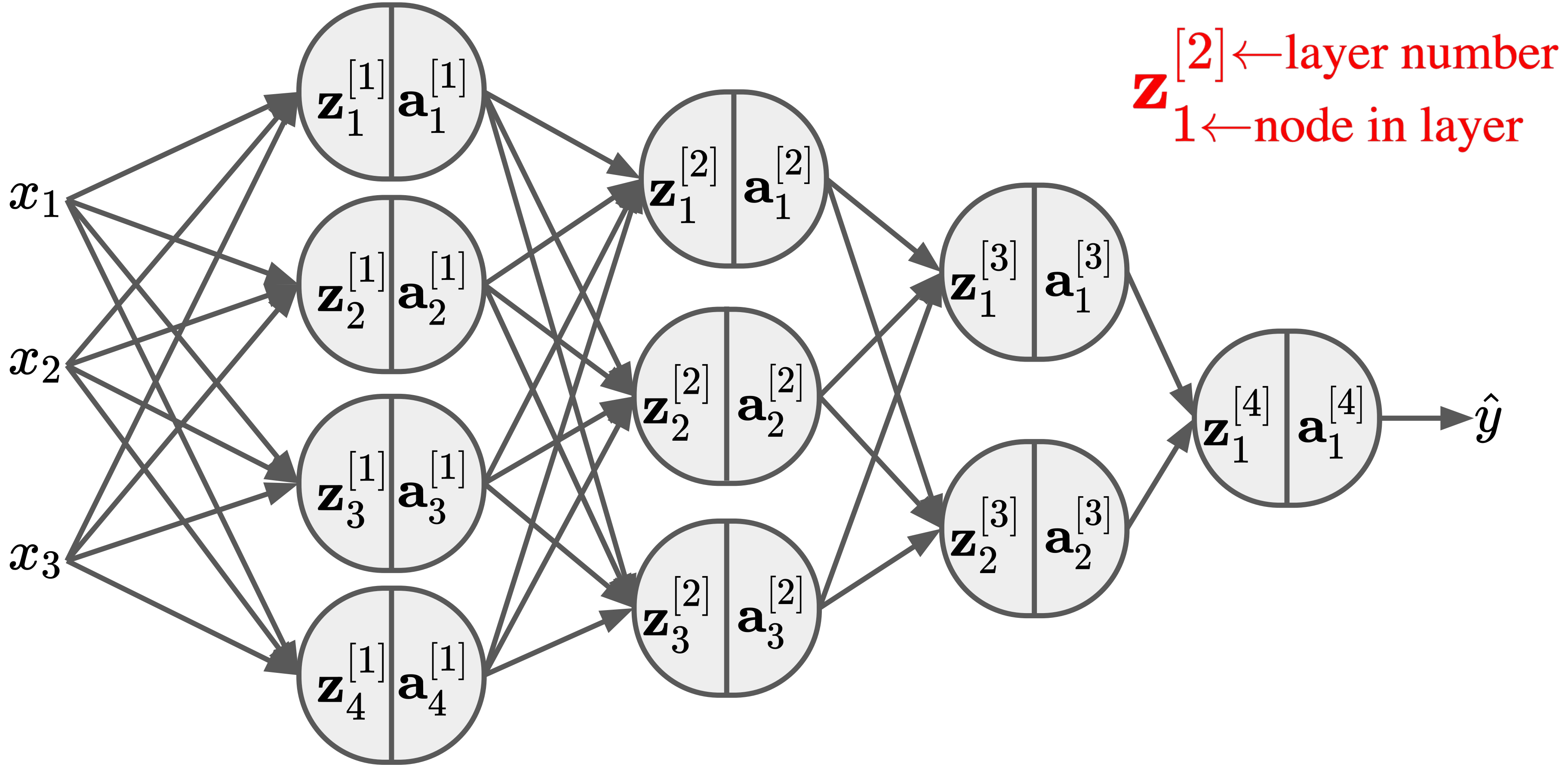
Recap: Anatomy of a neural network

Training a neural net revolves around the following objects:

- **Layers**, which are combined into a network
- **Input data**, which includes features and corresponding labels
- **Loss function**, which defines the feedback signal used for training
- **Optimizer**, which determines how the learning proceeds

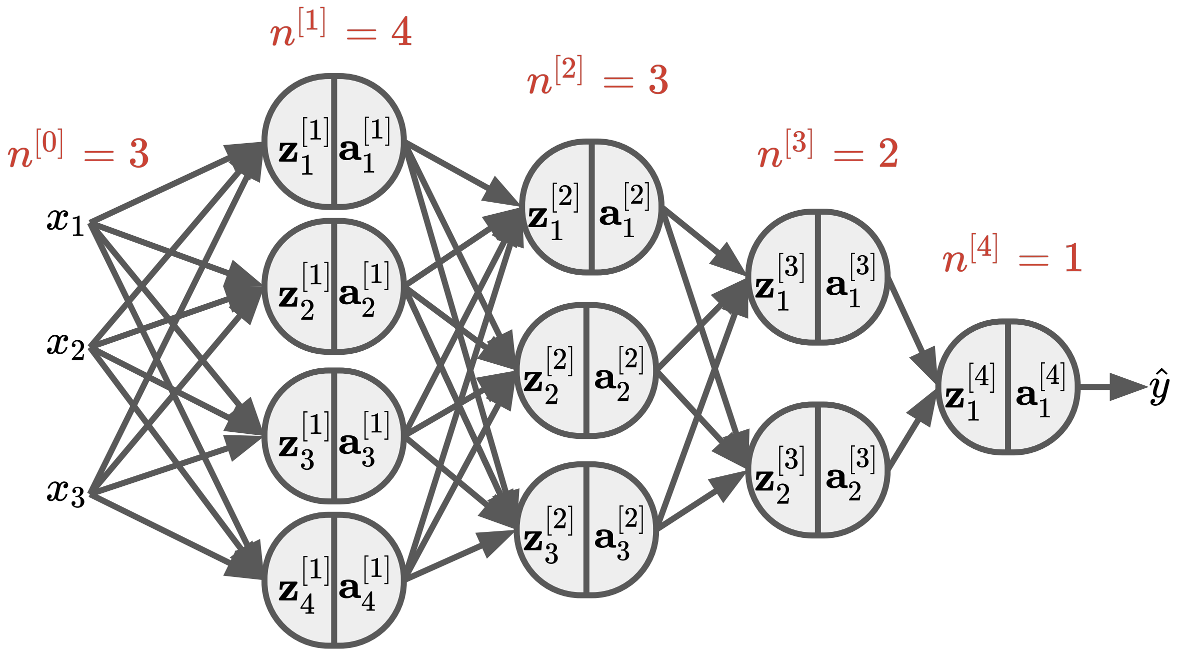


Layers: Building blocks of deep learning



x_1
x_2
x_3
$\mathbf{z}_1^{[1]}$ $\mathbf{a}_1^{[1]}$
$\mathbf{z}_2^{[1]}$ $\mathbf{a}_2^{[1]}$
$\mathbf{z}_3^{[1]}$ $\mathbf{a}_3^{[1]}$
$\mathbf{z}_4^{[1]}$ $\mathbf{a}_4^{[1]}$
$\mathbf{z}_1^{[2]}$ $\mathbf{a}_1^{[2]}$
$\mathbf{z}_2^{[2]}$ $\mathbf{a}_2^{[2]}$
$\mathbf{z}_3^{[2]}$ $\mathbf{a}_3^{[2]}$
$\mathbf{z}_1^{[3]}$ $\mathbf{a}_1^{[3]}$
$\mathbf{z}_2^{[3]}$ $\mathbf{a}_2^{[3]}$
$\mathbf{z}_1^{[4]}$ $\mathbf{a}_1^{[4]}$
$\hat{y}$
$\mathbf{z}_{1 \leftarrow \text{node in layer}}^{[2] \leftarrow \text{layer number}}$

Network of Neural Nets Layers



$n^{[0]} = 3$
$n^{[1]} = 4$
$n^{[2]} = 3$
$n^{[3]} = 2$
$n^{[4]} = 1$
x_1
x_2
x_3
$\mathbf{z}_1^{[1]}$ $\mathbf{a}_1^{[1]}$
$\mathbf{z}_2^{[1]}$ $\mathbf{a}_2^{[1]}$
$\mathbf{z}_3^{[1]}$ $\mathbf{a}_3^{[1]}$
$\mathbf{z}_4^{[1]}$ $\mathbf{a}_4^{[1]}$
$\mathbf{z}_1^{[2]}$ $\mathbf{a}_1^{[2]}$
$\mathbf{z}_2^{[2]}$ $\mathbf{a}_2^{[2]}$
$\mathbf{z}_3^{[2]}$ $\mathbf{a}_3^{[2]}$
$\mathbf{z}_1^{[3]}$ $\mathbf{a}_1^{[3]}$
$\mathbf{z}_2^{[3]}$ $\mathbf{a}_2^{[3]}$
$\mathbf{z}_1^{[4]}$ $\mathbf{a}_1^{[4]}$
$\hat{y}$

$$\mathbf{W}^{[l]} = (n^{[l]} \times n^{[l-1]})$$
$$\mathbf{b}^{[l]} = (n^{[l]} \times 1)$$
$$\mathbf{z}^{[l]} = (n^{[l]} \times 1)$$
$$\mathbf{a}^{[l]} = (n^{[l]} \times 1)$$

Vectorized Implementation

$$\mathbf{X} = \begin{bmatrix} \vert & \vert & & \vert \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ \vert & \vert & & \vert \end{bmatrix} \quad \mathbf{Z}^{[1]} = \begin{bmatrix} \vert & \vert & & \vert \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ \vert & \vert & & \vert \end{bmatrix} \quad \mathbf{A}^{[1]} = \begin{bmatrix} \vert & \vert & & \vert \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ \vert & \vert & & \vert \end{bmatrix}$$

$$\underset{\text{4xm}}{\mathbf{Z}^{[1]}} = \underset{\text{4x3}}{\mathbf{W}^{[1]}}\,\underset{\text{3xm}}{\mathbf{X}} + \underset{\text{4xm}}{\mathbf{b}^{[1]}} \qquad \underset{\text{4xm}}{\mathbf{A}^{[1]}} = \sigma(\underset{\text{4xm}}{\mathbf{Z}^{[1]}})$$

Vectorized Implementation

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{Z}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{z}^{[1](1)} & \mathbf{z}^{[1](2)} & \cdots & \mathbf{z}^{[1](m)} \\ | & | & & | \end{bmatrix} \quad \mathbf{A}^{[1]} = \begin{bmatrix} | & | & & | \\ \mathbf{a}^{[1](1)} & \mathbf{a}^{[1](2)} & \cdots & \mathbf{a}^{[1](m)} \\ | & | & & | \end{bmatrix}$$

$$\mathbf{W}^{[l]} = (n^{[l]} \times n^{[l-1]})$$

$$\mathbf{Z}^{[l]} = (n^{[l]} \times m)$$

$$\mathbf{A}^{[l]} = (n^{[l]} \times m)$$

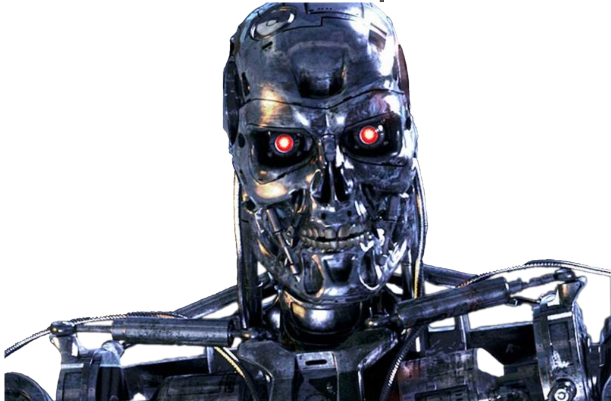
$$\mathbf{b}^{[l]} = (n^{[l]} \times m)$$

Recap: Loss functions and optimizers

Loss (objective) function: the quantity that will be minimized during the training process. It represents a measure of success for the task at hand

Optimizer: determines how the network will be updated based on the loss function (ie. Stochastic Gradient Descent)

Choosing the right objective function is extremely important! If the objective function doesn't fully correlate with the task at hand, your network will end up doing things you may not have wanted...



Training Deep Neural Nets

To tackle very complex problems, we need to train a much deeper network with tens of layers, hundreds of neurons. A few issue with deep neural nets includes:

- Facing the tricky “vanishing/exploding gradients” problem
- Training will be extremely slow
- Thousands of parameters risking overfitting

After today, you will able to train a very deep nets: welcome to **Deep Learning!**

The vanishing gradient problem

Training a DNN is not a walk in the park. Backpropagation works by passing the error gradient back and forth among input, output, and hidden layers. It uses gradient to update the parameters

Gradients get **smaller and smaller** to which point they leave the connection weights virtually **unchanged** → vanishing gradient problem.

Exploding Gradient Problem

The opposite of vanishing gradient can also happen

Gradients grow bigger and bigger → many layers got **insanely large** weight updates, and the network becomes unstable and diverged

A reason why deep neural networks were abandoned for a LONG time!
How to tuning the network?

Today: A Practical Guide to DNN configurations

1. **Initialization:** how to initialize the weights so that they do not saturate?
2. **Activation:** how to solve the vanishing/exploding gradient problem?
3. **Normalization:** how to get the model to learn the optimal scale?
4. **Regularization:** is it just using L1 and L2, or is it something more?
5. **Optimization:** when gradient descent was too slow or not good enough?
6. **Learning Rate**: what if convergence is too slow or sub-optimal?

1. Initialization

Xavier Initialization

We don’t want the signal to die out nor to explode and saturate.

To keep it **flow properly,** Xavier Glorot and his advisor Yoshua Bengio argue that “*we need the variance of the outputs of each layer to be* ***equal*** *to the variance of its inputs*” → Xavier Initialization for Logistic Activation Function

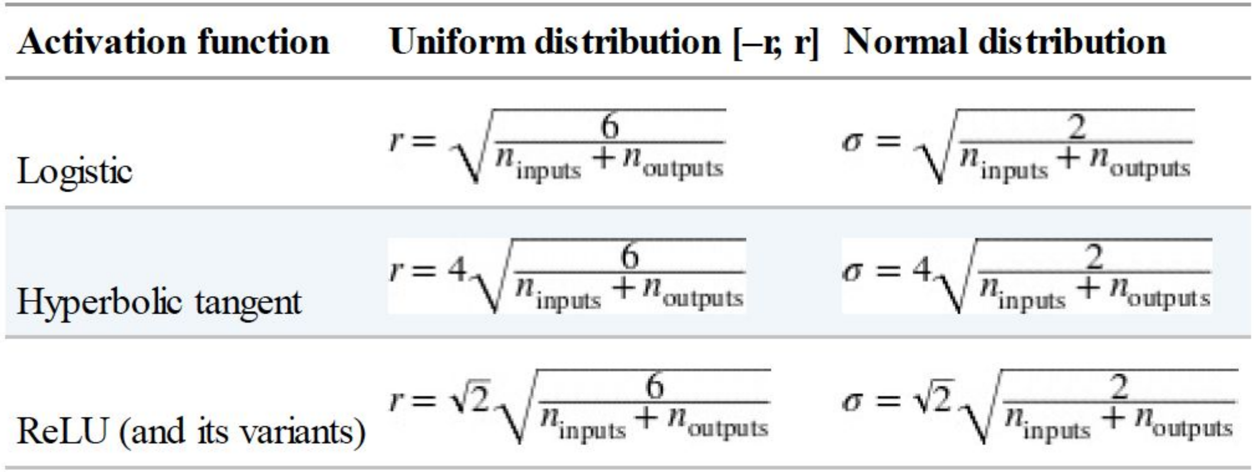
This initialization strategy led to the current success of Deep Learning.

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

He Initialization

Similar to Xavior’s, but for different activation functions



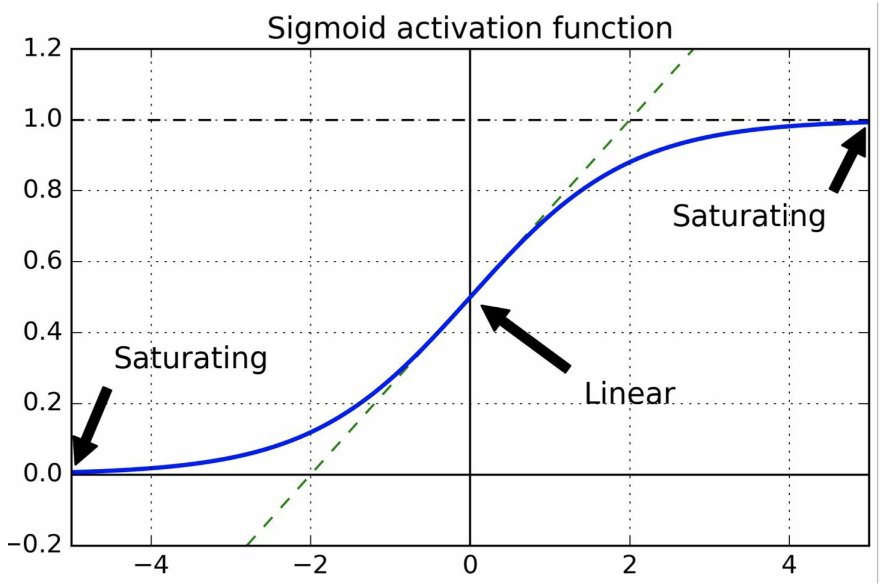
Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

2. Activation Function

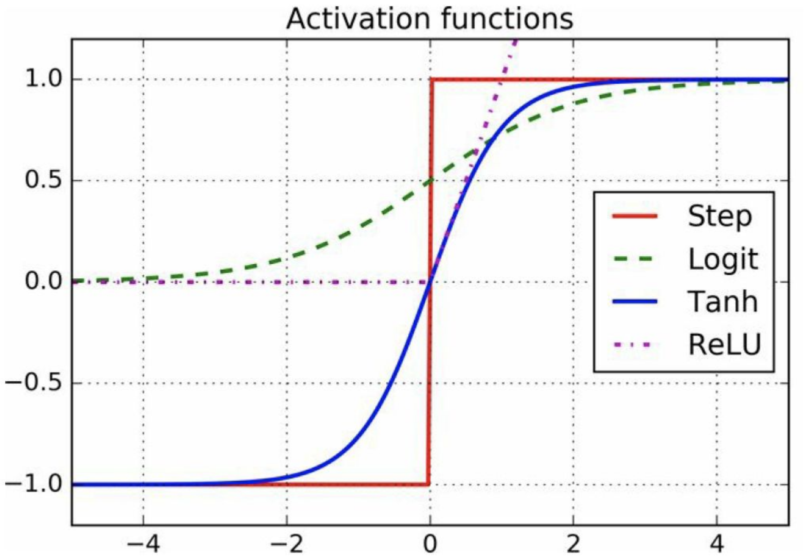
Activation Function Choices

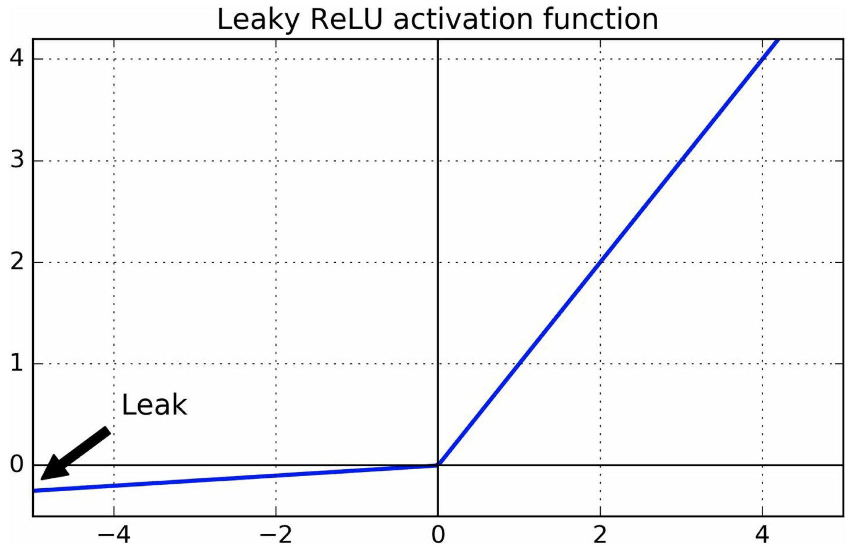
- Poor choice of activation function can lead to vanishing/exploding gradient
- Mother Nature chooses to use roughly sigmoid activation function in biological neurons, but it turns out that other functions (eg. ReLU) behave much faster and better.



ReLU Activation

- Fast to compute, but suffer from **dying**: meaning not outputting anything but 0
- To solve this, you can use a variant of ReLU called leaky ReLU

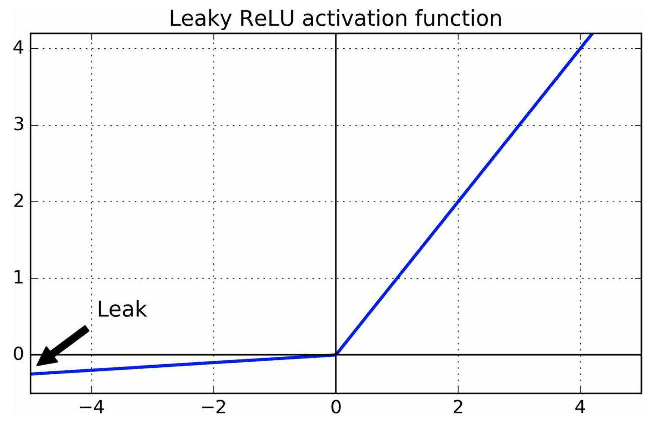


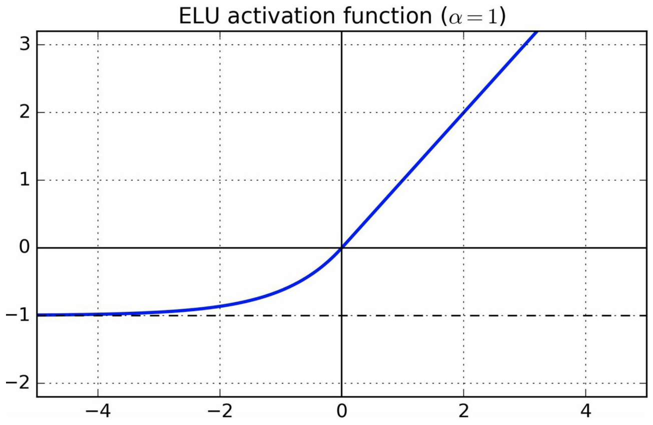


ELU (Exponential Linear Unit)

Outperform ReLU on faster convergence and accuracy (Clevert et al 2015)

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & if z \geq 0 \end{cases}$$





Why ELU is better than leaky ReLU?

```
layer = keras.layers.Dense(10, activation="selu",
```

3. Normalization

Batch Normalization

He Initialization and ELU reduce vanishing/exploding gradient problems at the **beginning** of training, but does not guarantee they won’t come back **during** training

Sergey Ioffe and Christian Szegedy (in their 2015 paper) address this by a technique called Batch Normalization (BN)

This adds an operation **before** activation function: simply zero-centering and normalizing the inputs, then **scaling** and **shifting** the results → optimal scale.

Batch Normalization

Learn 4 parameters: scale, offset, mean, and standard deviation

$$\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

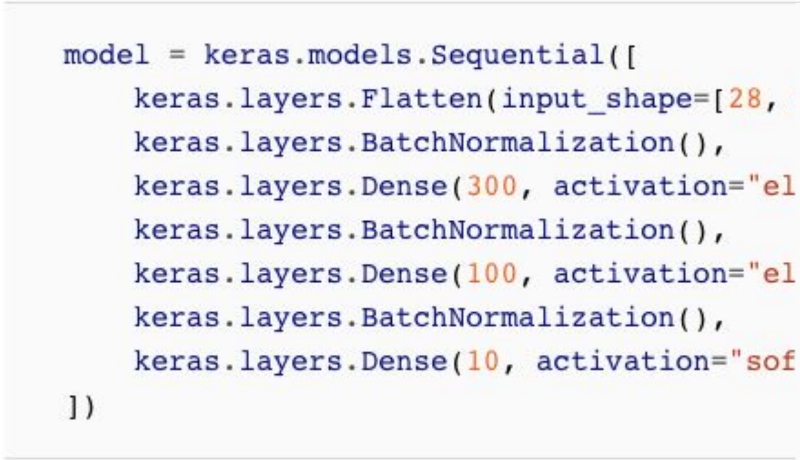
$$\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

$$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \varepsilon}}$$

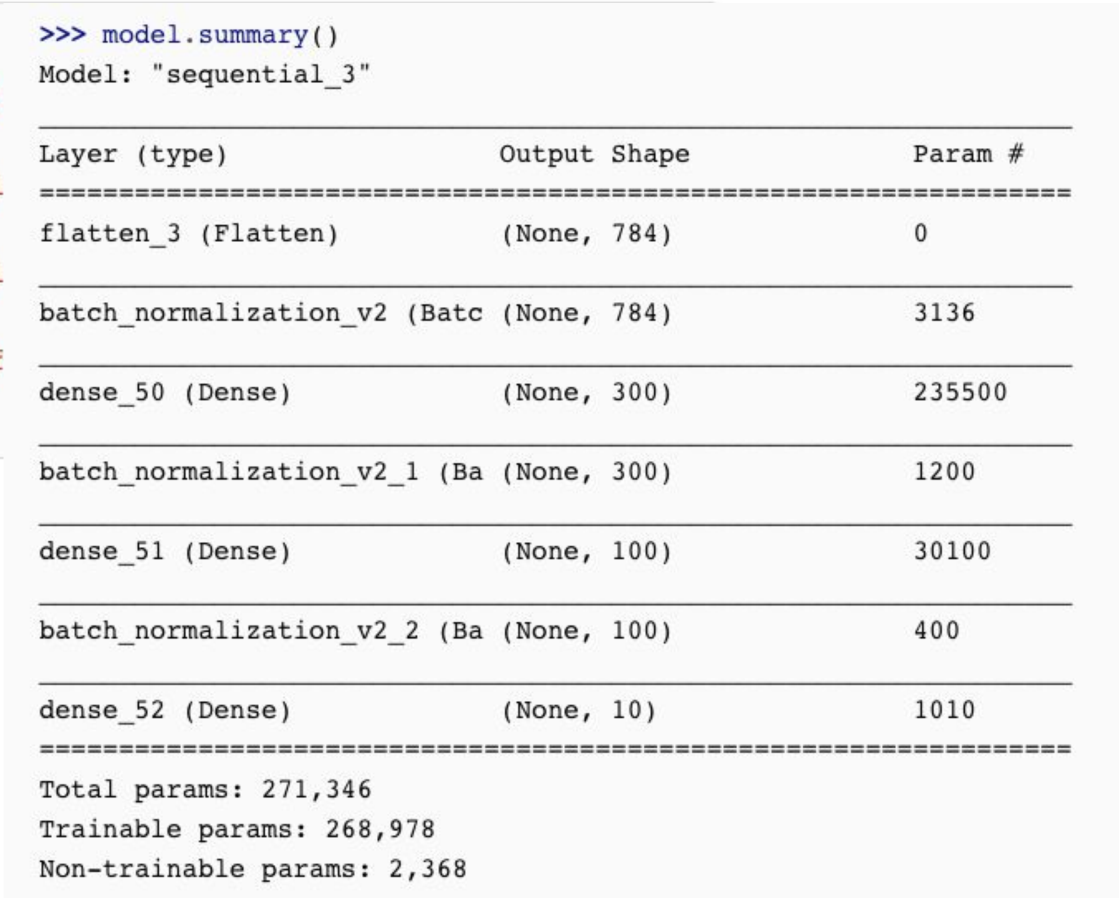
$$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

- μ_B is the empirical mean, evaluated over the whole mini-batch B.
- σ_B is the empirical standard deviation, also evaluated over the whole mini-batch.
- m_B is the number of instances in the mini-batch.
- $\mathbf{X}^{(i)}$ is the zero-centered and normalized input.
- γ is the scaling parameter for the layer.
- β is the shifting parameter (offset) for the layer.
- ϵ is a tiny number to avoid division by zero (typically 10^{-3}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a scaled and shifted version of the inputs.

Batch Normalization Implementation



```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28,
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="el
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="el
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="sof
])
```



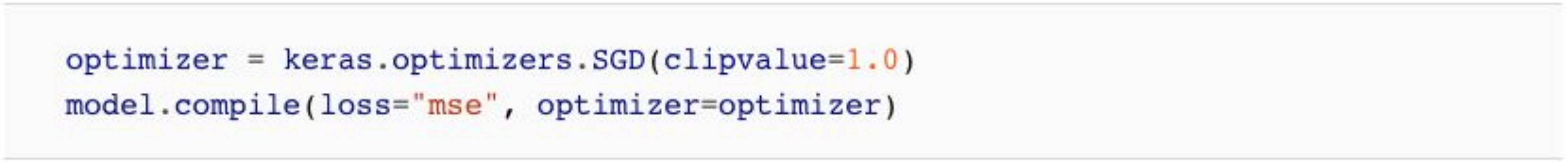
```
>>> model.summary()
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Gradient Clipping

Use as an alternative to Batch Normalization

Quick and dirty: simply clip the gradients during backpropagation so that they can never exceed some threshold.

Often use in Recurrent Neural Nets (RNNs)



```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

4. Regularization

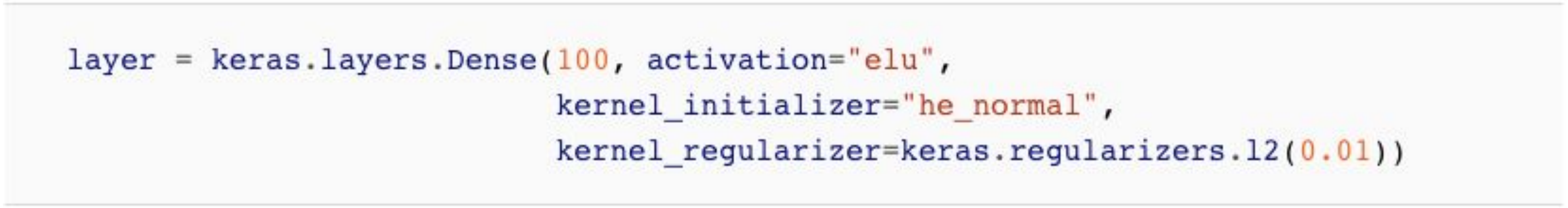
Previous Regularization

Avoid overfitting with thousands of parameters

We already know some methods so far:

- Early Stopping: interrupt training when its performance on validation set starts dropping
- L1 and L2 Regularization: constraints the neural network’s connection weights

But there is ***a more effective*** regularization technique for deep neural networks



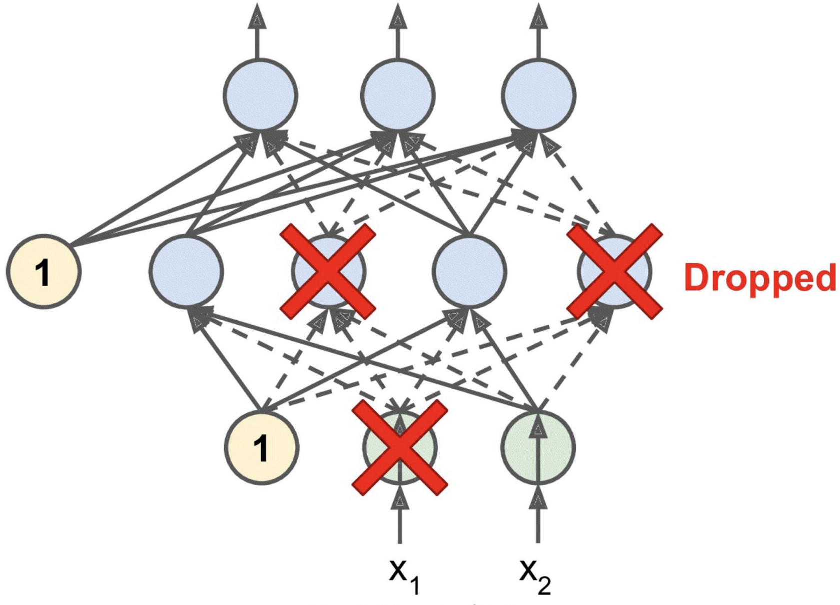
```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Dropout

Proposed by Hinton in his 2012 paper, and it's very simple:

At every training step, every neuron has a probability `p` of being temporarily "dropped out" (entirely ignore until the next step). The hyperparameter p is called the dropout rate (typically set at 50%)

`(1-p)` is the keep probability

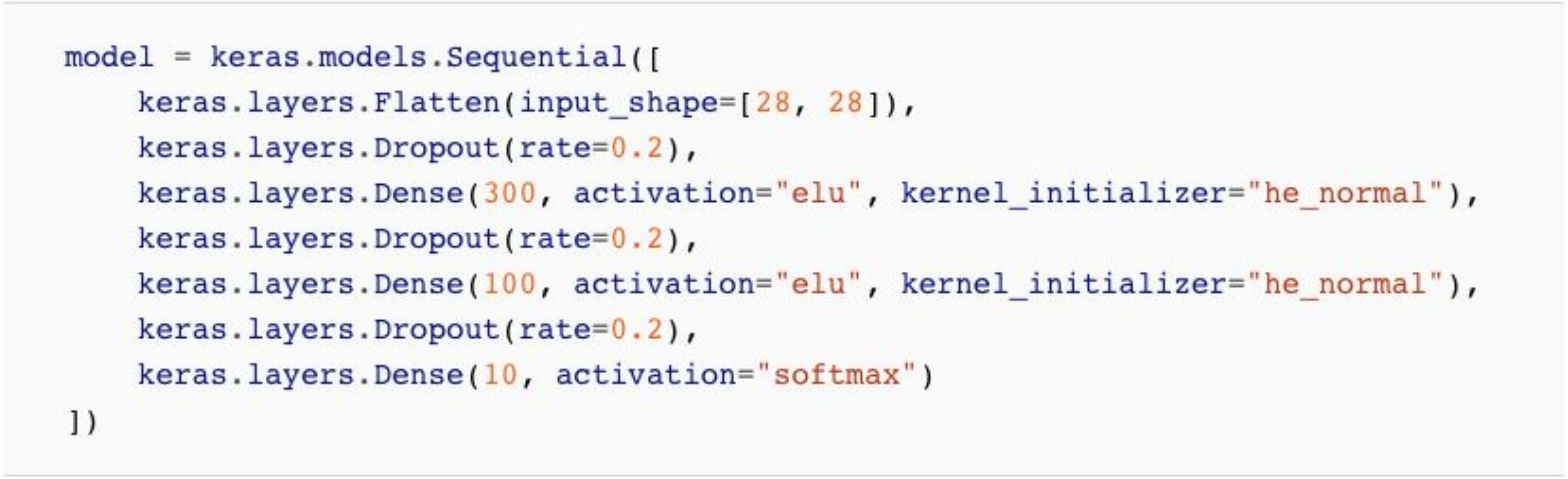


Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work?

Justification for dropout

- Company needs to adapt its organization: cannot rely on any single person
- Employees would have to learn to cooperate with many coworkers
- If one person quit, it wouldn’t make much a difference

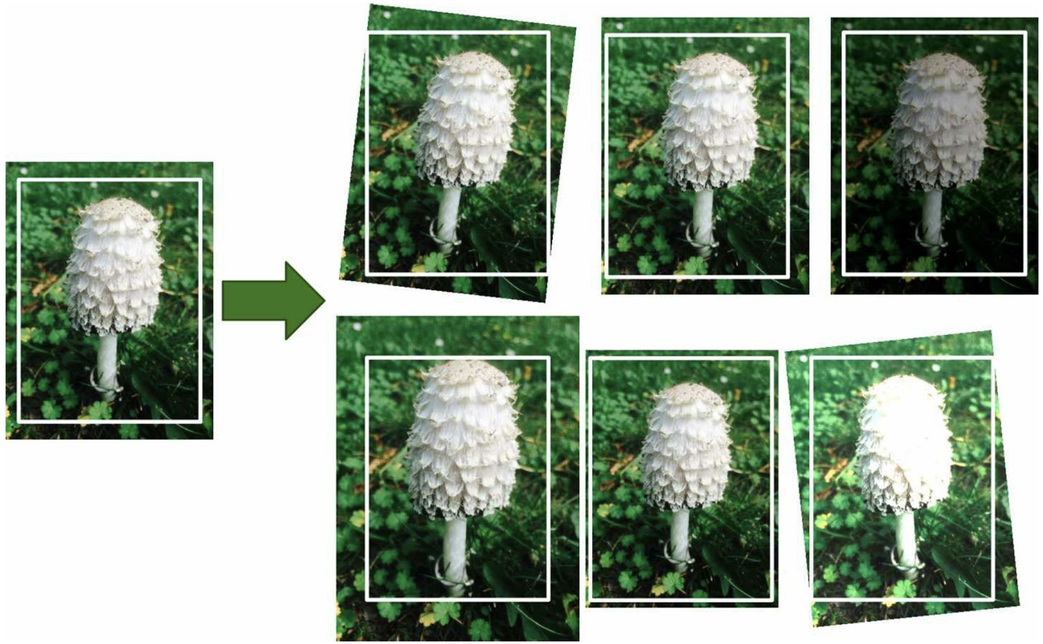
It’s unclear whether it works for **real** company, but it works for a more robust network



```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Data Augmentation

Generating new training samples from existing one to artificially boost the size of training set → reduce overfitting.



5. Optimization

Optimization Functions

Training can be extremely slow → huge **speed boost** comes from using a faster optimizer than the **good-old Gradient Descent**.

What are the choices?

- Momentum Optimization
- AdaGrad
- RMSProp
- Adam (Adaptive Moment Estimation)
    - combines ideas of the above

Momentum Optimization

Gradient Descent take small regular steps down the slope → slow

$$\theta \rightarrow \theta - \alpha \nabla_\theta J(\theta)$$

Imagine a bowling ball rolling down a slope on a smooth surface → accelerate

Momentum Optimization uses gradient as **acceleration**, not as a speed.

$$\mathbf{m} \rightarrow \beta \mathbf{m} - \alpha \nabla_\theta J(\theta)$$
$$\theta \rightarrow \theta + \mathbf{m}$$

It may help roll past local optima, but does this cause any issue?

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

AdaGrad

Consider a elongated bowl, GD may start quickly by going down the steepest slope, but will slowly go down the bottom of the valley

AdaGrad detects this early and correct its direction to point a bit more toward global optimum (adaptive learning rate)

It achieves this by accumulating the squares of the gradients vector along the steepest dimensions

$$\mathbf{s} \to \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$\theta \to \theta - \alpha \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s}}$$

RMSProp

AdaGrad slows down too fast and may not converge on global optimum

RMSProp fixes this by accumulating only the gradient from most recent iteration by using exponential decay β (set at 0.9) if the first step.

$$\mathbf{s} \to \beta\mathbf{s} + (1-\beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$\theta \to \theta - \alpha\nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s}}$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam (Adaptive Moment Estimation)

Combines ideas of Momentum optimization and RMSProp: Like Momentum, it keeps track of an exponentially decaying average of past gradients

Like RMSProp, it also keeps track of an exponentially decaying average of past square gradients

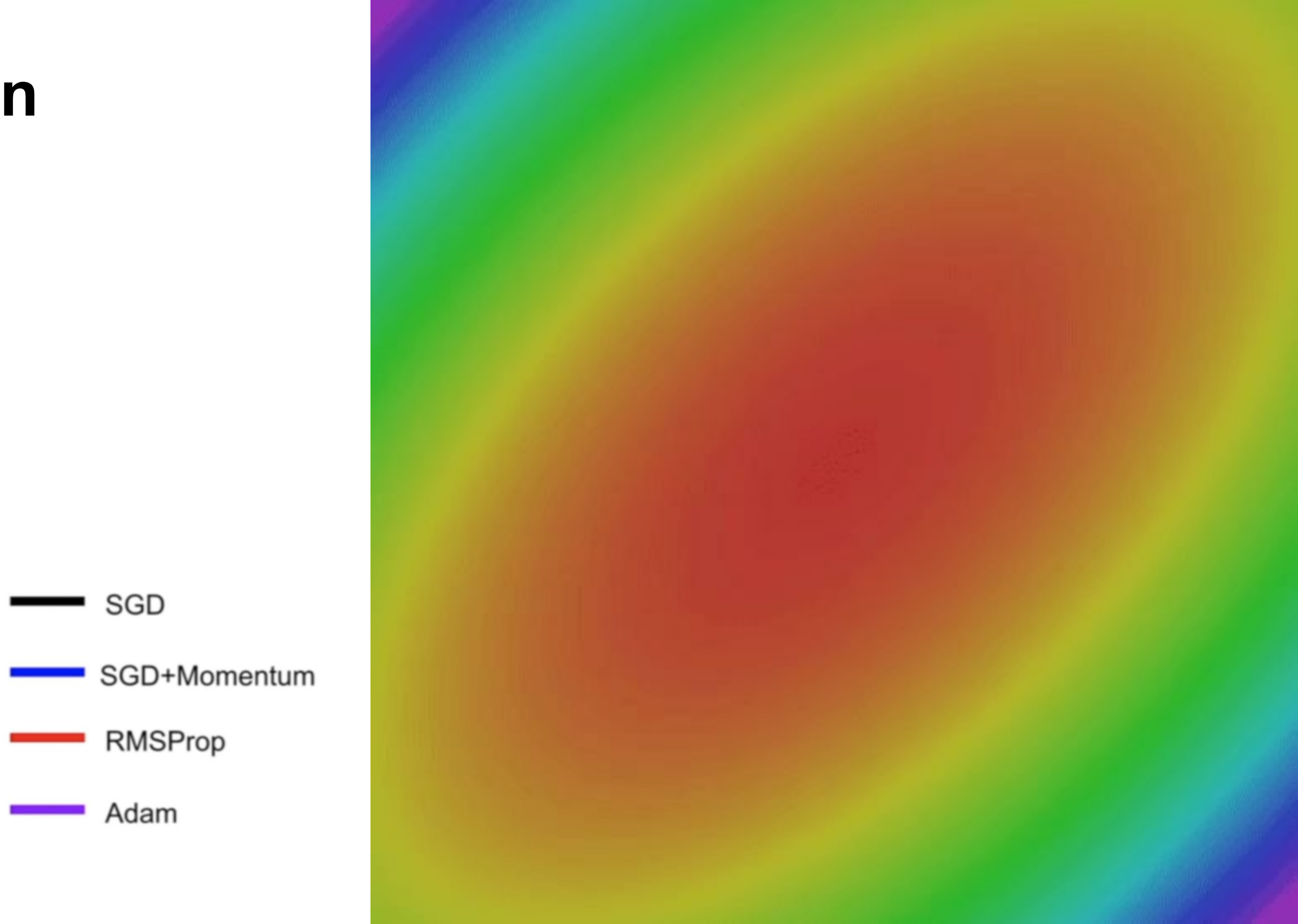
$$\mathbf{m} \to \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J(\theta)$$

$$\mathbf{s} \to \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

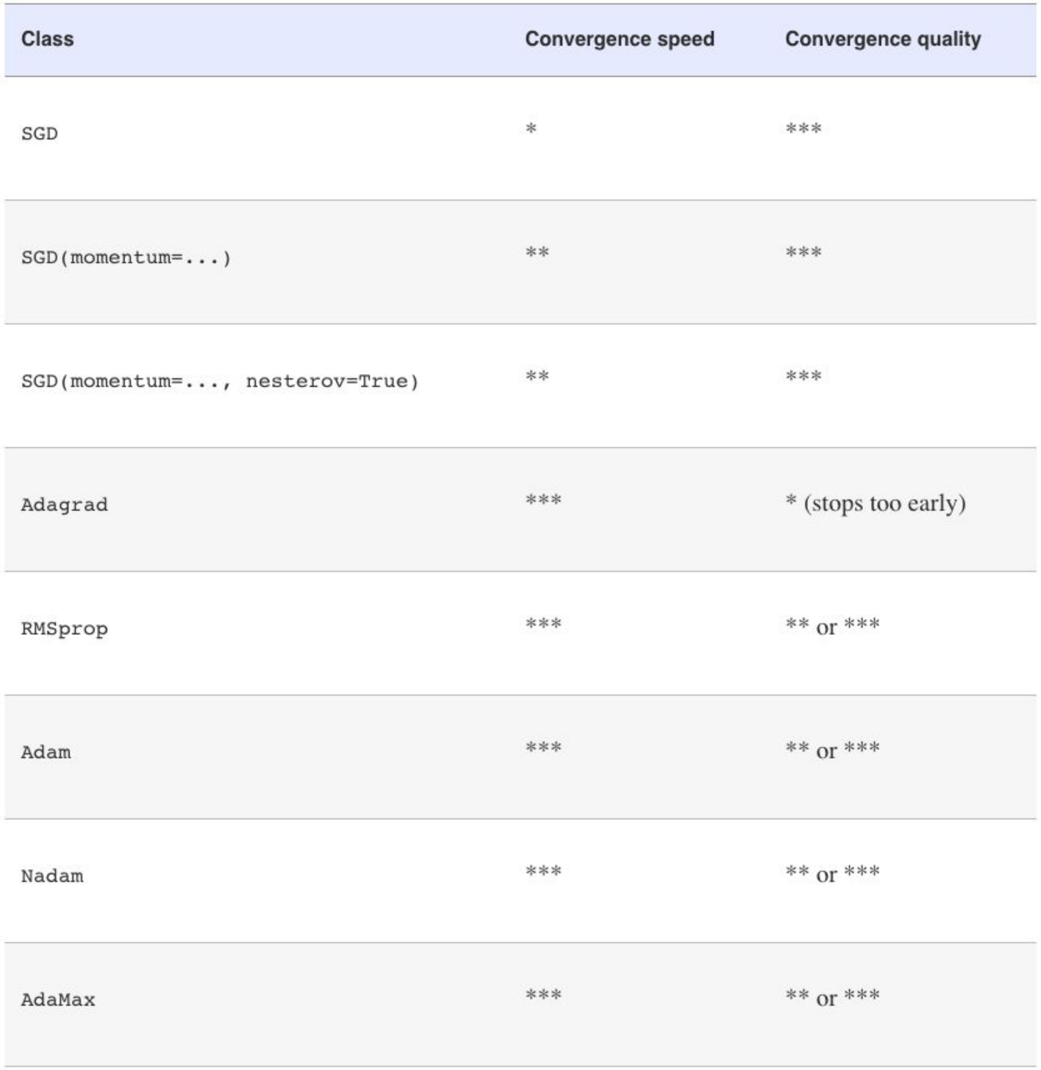
$$\theta \to \theta - \alpha \mathbf{m} \oslash \sqrt{\mathbf{s}}$$

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Animation



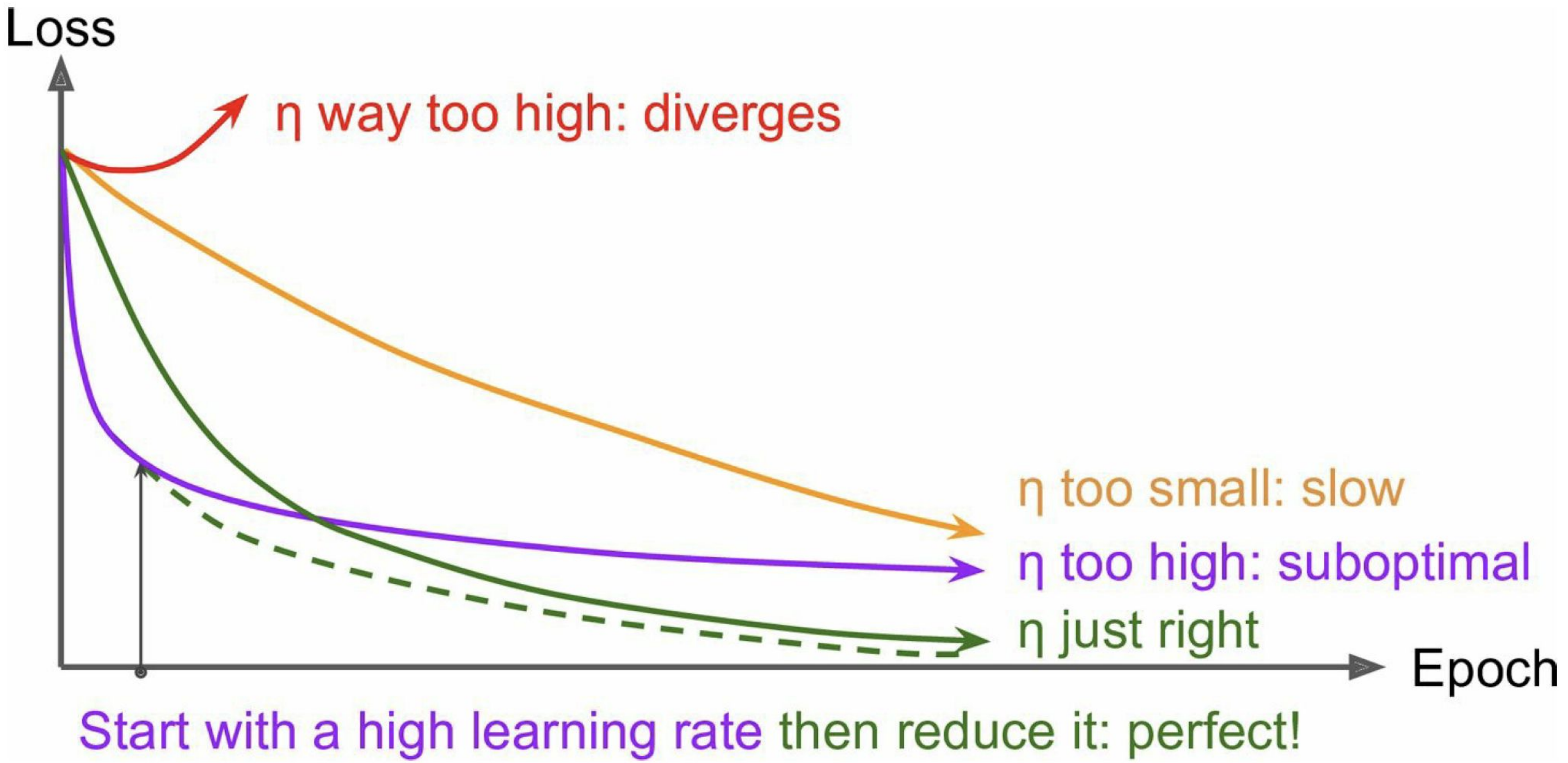
Optimizer Comparison



Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

6. Learning Rate Scheduling

Loss vs. Iteration Epoch



Most common strategies for learning schedule

Predetermined piecewise constant learning rate

For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling

Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling

Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\ 10^{-t/r}$. This works great, but it requires tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

Power scheduling

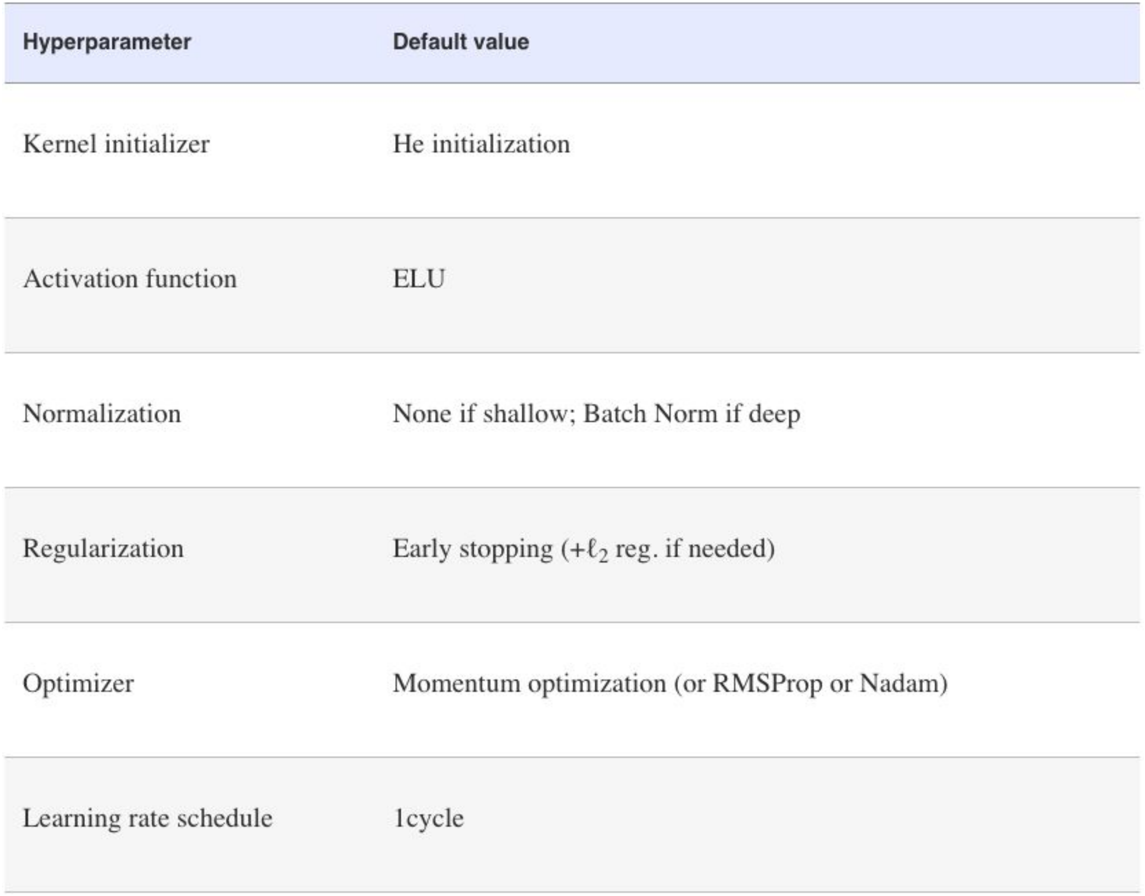
Set the learning rate to $\eta(t) = \eta_0\ (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

Implementing the Learning Rate

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Summary: the Practical Guidelines



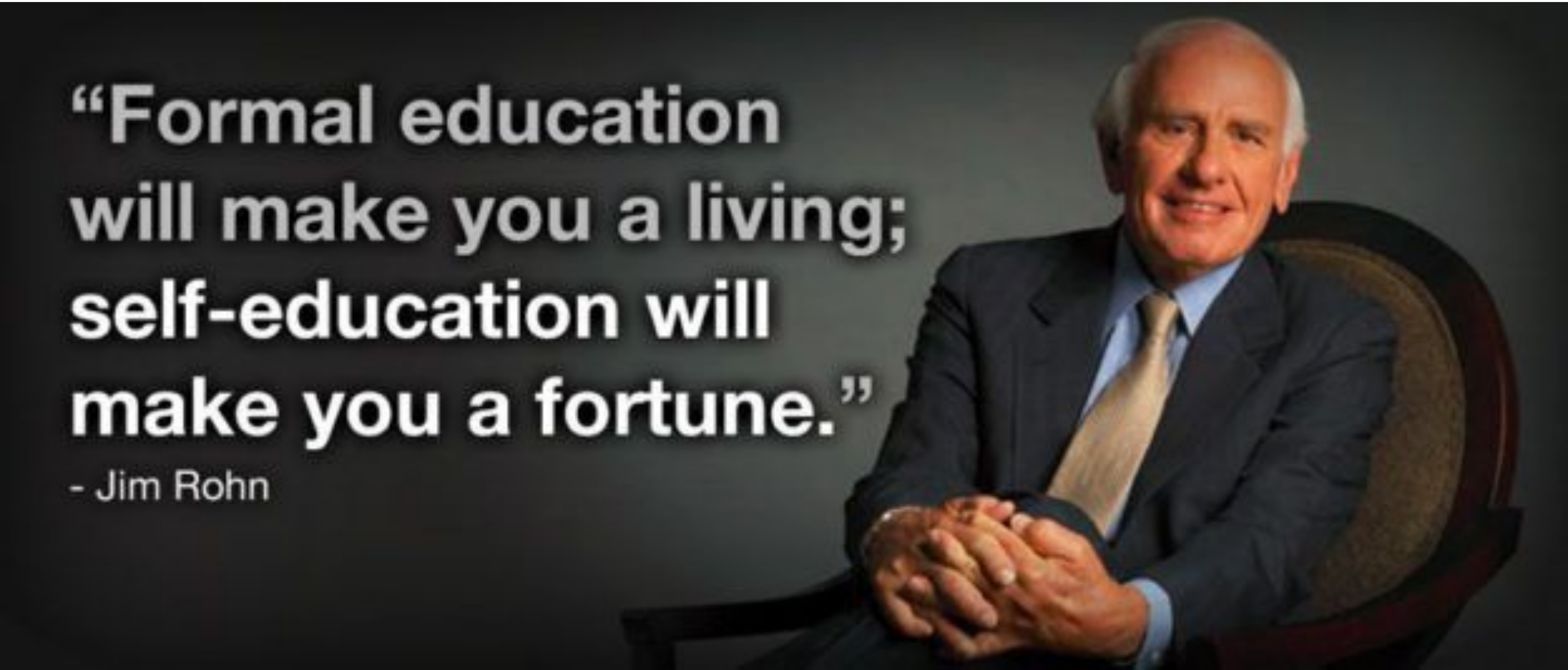
Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Future Studies of Deep Learning

We barely scratch the surface of Deep Learning, there are still a lot more:

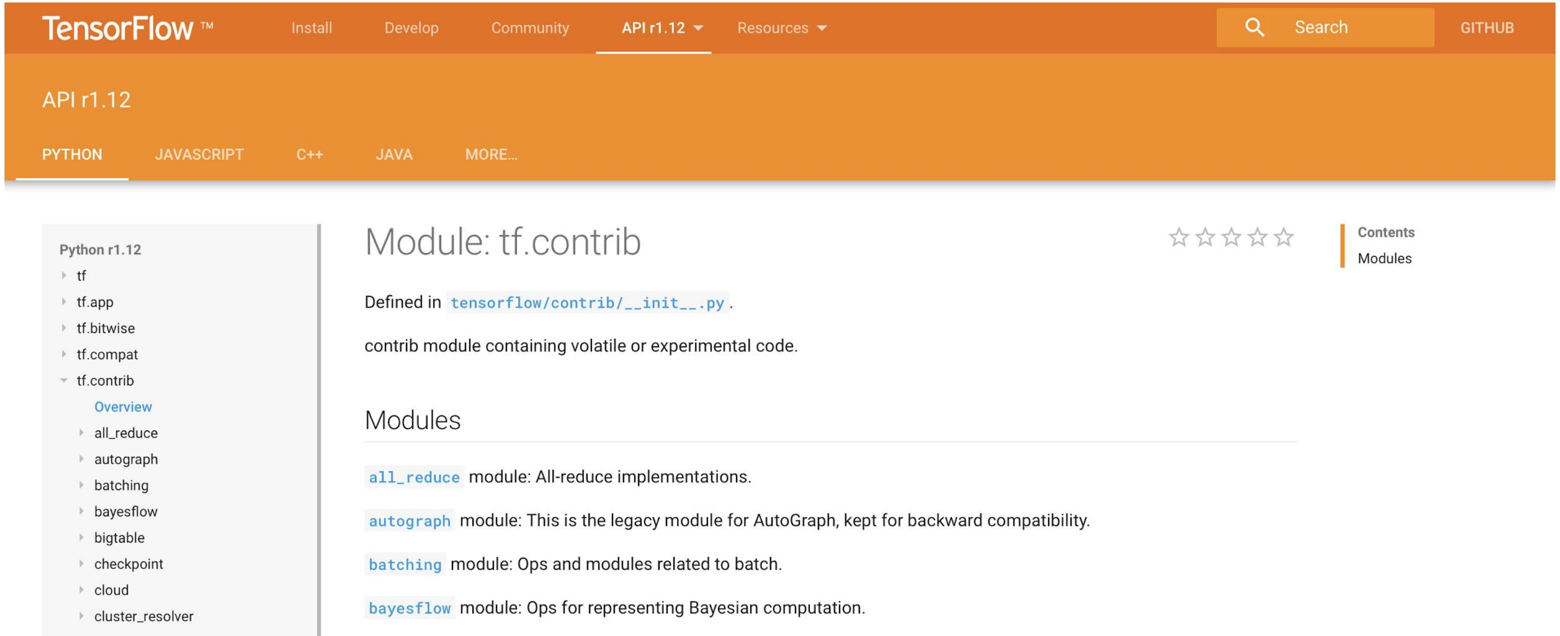
- Convolutional Neural Nets (CNNs)
- Recurrent Neural Nets (RNNs)
- Autoencoders and GANs

Don’t stop (Deep) Learning!



A note on the implementation details of DNN

In this lecture, we are mostly cover the concepts without diving too much into the implementation details. However, in the coding assignment, you will use what we are about to discuss to find the appropriate TensorFlow API.



Module: tf.contrib

Defined in `tensorflow/contrib/__init__.py`.

contrib module containing volatile or experimental code.

Modules

`all_reduce` module: All-reduce implementations.

`autograph` module: This is the legacy module for AutoGraph, kept for backward compatibility.

`batching` module: Ops and modules related to batch.

`bayesflow` module: Ops for representing Bayesian computation.