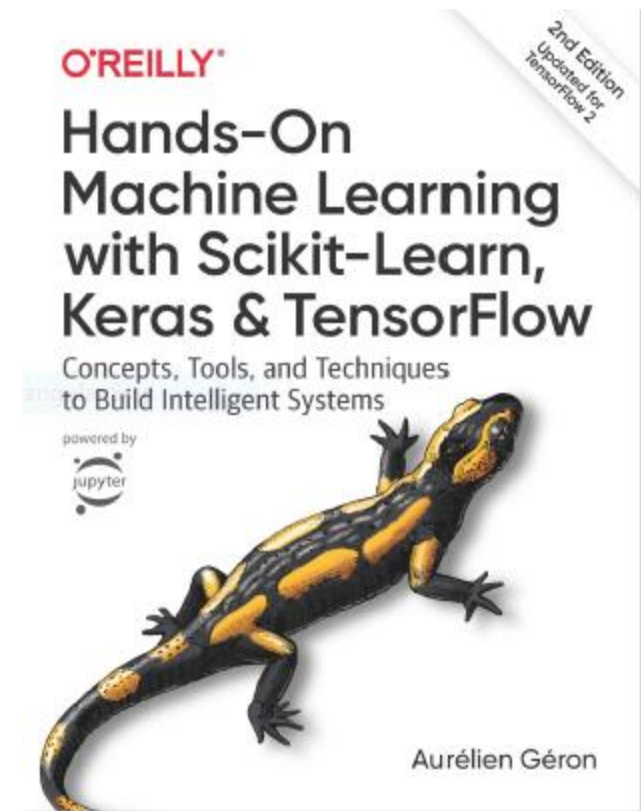


Deep Neural Networks

Prof. Gheith Abandah

Reference

- Chapter 11: **Training Deep Neural Networks**



- Aurélien Géron, **Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow**, O'Reilly, 2nd Edition, 2019
 - Material: <https://github.com/ageron/handson-ml2>

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

1. Introduction

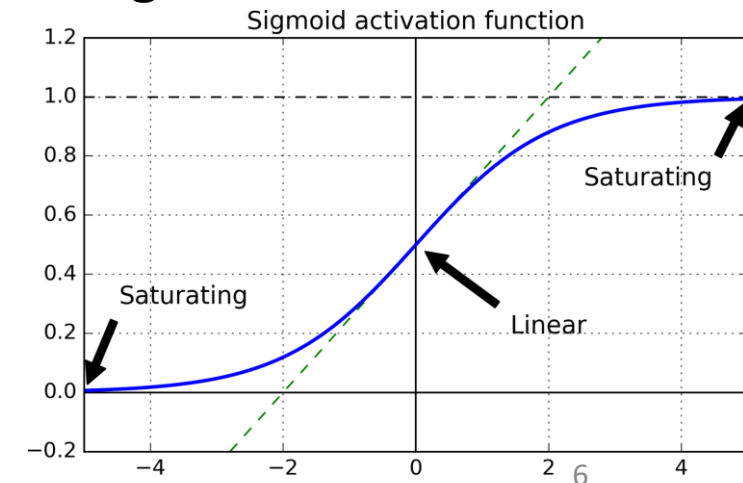
- Deep neural networks can solve **complex problems** and provide **end-to-end** solutions.
- When you train a deep network, you may face the following **problems**:
 - **Vanishing** or **exploding gradients**: The gradients grow smaller and smaller, or larger and larger.
 - **Not enough data**
 - **Long training time**
 - **Overfitting**

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

2. Vanishing/Exploding Gradients Problems

- **Vanishing Problem:** In the backpropagation algorithm, gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
 - Lower layers' connections are left unchanged.
- **Exploding Problem:** the gradients can grow bigger and bigger.
 - Layers get very large weight updates and the algorithm diverges.
- **Main Reasons:** Using activation functions (logistic sigmoid) and weight initialization (normal distribution with 0-mean and 1-standard deviation).



2.1 Glorot and He Initialization

- **Glorot and Bengio**: In order for the signal not to die out, nor to explode and saturate, the variance of the outputs of each layer should be equal to the variance of its inputs.
- **Solution**: the connection weights of each layer must be initialized randomly as follows:

Normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{fan_{avg}}$

Or a uniform distribution between $-r$ and $+r$, with $r = \sqrt{\frac{3}{fan_{avg}}}$

$$fan_{avg} = (fan_{in} + fan_{out})/2.$$

2.1 Glorot and He Initialization

- **Recommended** initialization parameters for each type of activation function.

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

- For the uniform distribution, use $r = \sqrt{3\sigma^2}$
- Keras uses **Glorot initialization** with a **uniform** distribution.

2.1 Glorot and He Initialization

- To change it to **He initialization**:

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

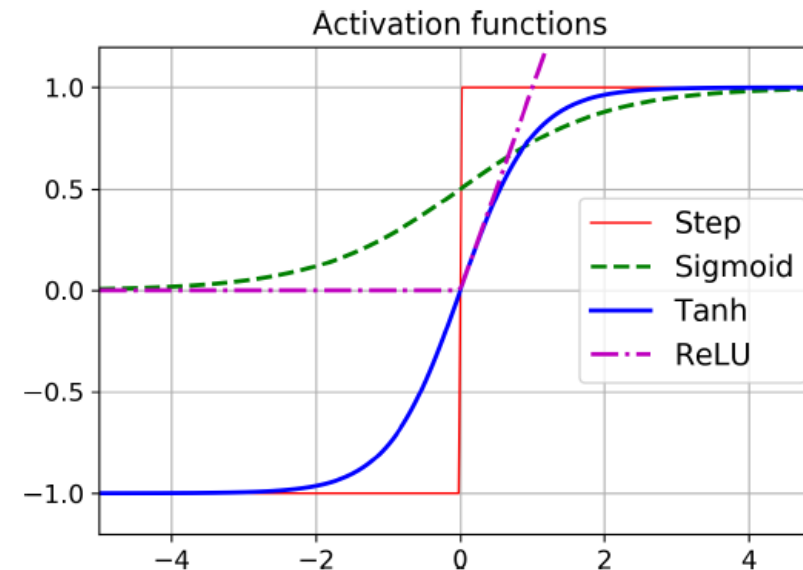
- **He initialization** with a **uniform** distribution but based on **fan_{avg}**:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,  
    mode='fan_avg', distribution='uniform')
```

```
keras.layers.Dense(10, activation="sigmoid",  
    kernel_initializer=he_avg_init)
```

2.2 Nonsaturating Activation Functions

- **Step** does not work with the back propagation algorithm.
- **ReLU** is better than **sigmoid** because it does not saturate for positive values and is fast.
- **Dying ReLUs**: A neuron dies when its input is negative for all training instances.

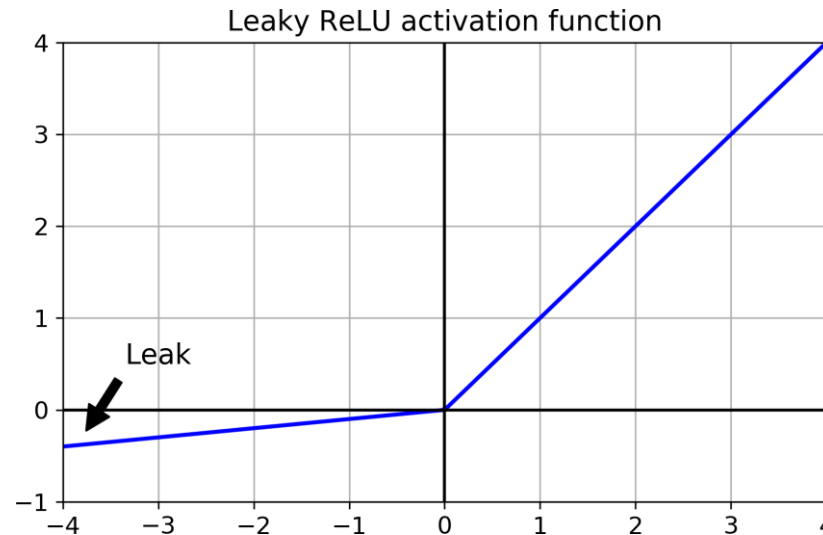


2.2 Nonsaturating Activation Functions

- **Leaky ReLU** performs better than ReLU.

$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$$

- α between 0.01 and 0.3

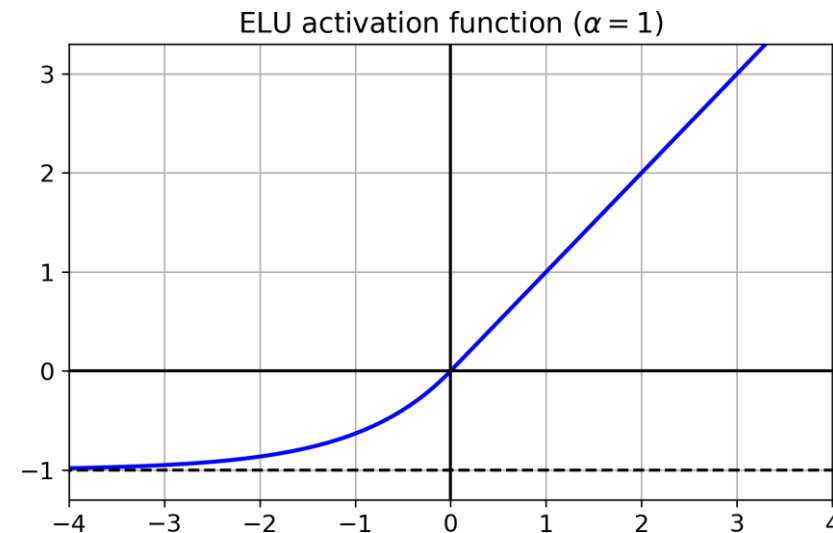


```
model = keras.models.Sequential([  
    ...  
    keras.layers.Dense(10, kernel_initializer="he_normal"),  
    keras.layers.LeakyReLU(alpha=0.2), # added as a layer  
    ...  
])
```

2.2 Nonsaturating Activation Functions

- **Exponential linear unit (ELU)** also performs better than ReLU but is slower.
- **Scaled ELU (SELU)** performs best with dense and CNN, but must scale inputs and use `lecun_normal`.

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



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

2.2 Nonsaturating Activation Functions

- **Summary:**

- SELU > ELU > leaky ReLU > ReLU > tanh > logistic
- If you cannot use SELU, use ELU.
- For fast response, use leaky ReLU or ReLU.

2.3 Batch Normalization

- The techniques in §2.1 and §2.2 can significantly reduce the vanishing/exploding gradients problems at the **beginning of training**, but don't guarantee that they won't **come back during training**.
- **Batch Normalization (BN)** zero-centers and normalizes each layer input using statistics from the mini batch (> 30).
- **Other benefits**: Works even without §2.1 and §2.2, allows using larger LR, and have regularization effect.

2.3 Batch Normalization

- Implementing batch normalization with Keras is easy.

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

2.4 Gradient Clipping

- Mitigates the exploding gradients problem by **clipping the gradients** during backpropagation so that they never exceed some threshold.
- Use it when you observe that the gradients are exploding during training. You can **track the size of the gradients** using TensorBoard.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

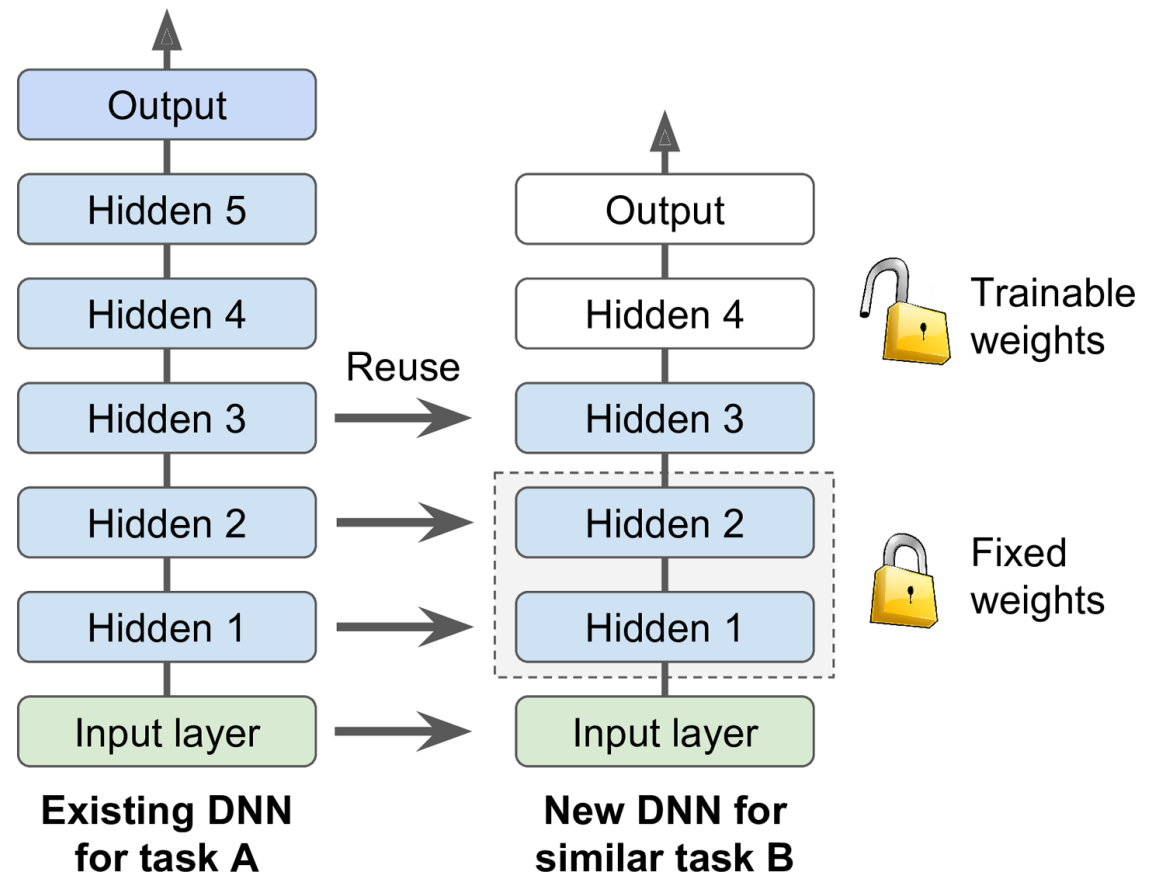
```
model.compile(loss="mse", optimizer=optimizer)
```


Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

3. Reusing Pretrained Layers

- **Transfer Learning:** Using one NN developed for a certain task to solve another task.
- Useful to **shorten training time** or with **small datasets**.



Transfer Learning with Keras

```
# Load the ready model
model_A = keras.models.load_model("my_model_A.h5")
# Create a new model using all but the last layer
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
# Freeze loaded layers then compile
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False

model_B_on_A.compile(loss="binary_crossentropy",
                    optimizer="sgd", metrics=["accuracy"])
```

Transfer Learning with Keras

```
# Train the model for a few epochs
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))
# Unfreeze loaded layers
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = True
# Compile with small learning rate (default = 1e-2)
optimizer = keras.optimizers.SGD(lr=1e-4)
model_B_on_A.compile(loss="binary_crossentropy",
                    optimizer=optimizer, metrics=["accuracy"])
```

Transfer Learning with Keras

```
# Train the model for more epochs
```

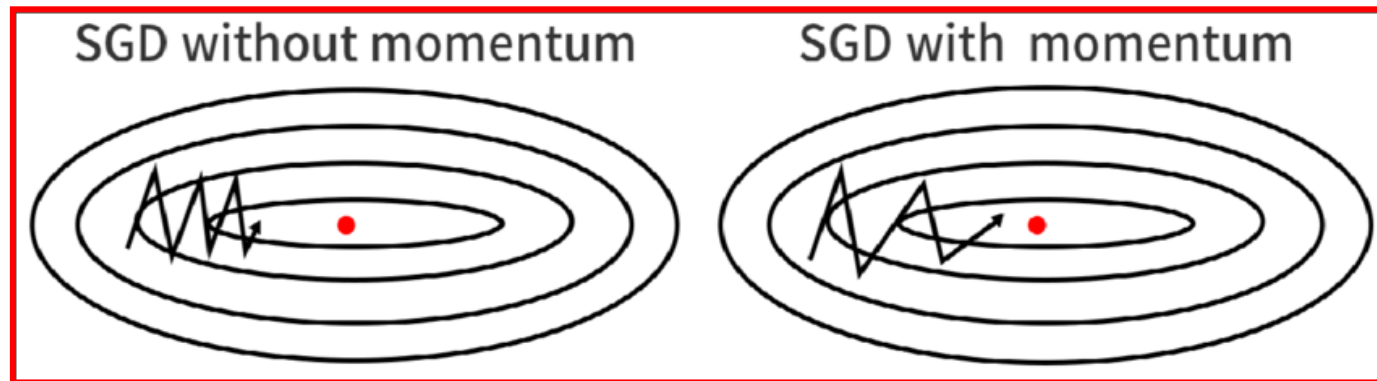
```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                           validation_data=(X_valid_B, y_valid_B))
```

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

4. Faster Optimizers

- The SGD optimizer can be made faster using **momentum optimization**



$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$$

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

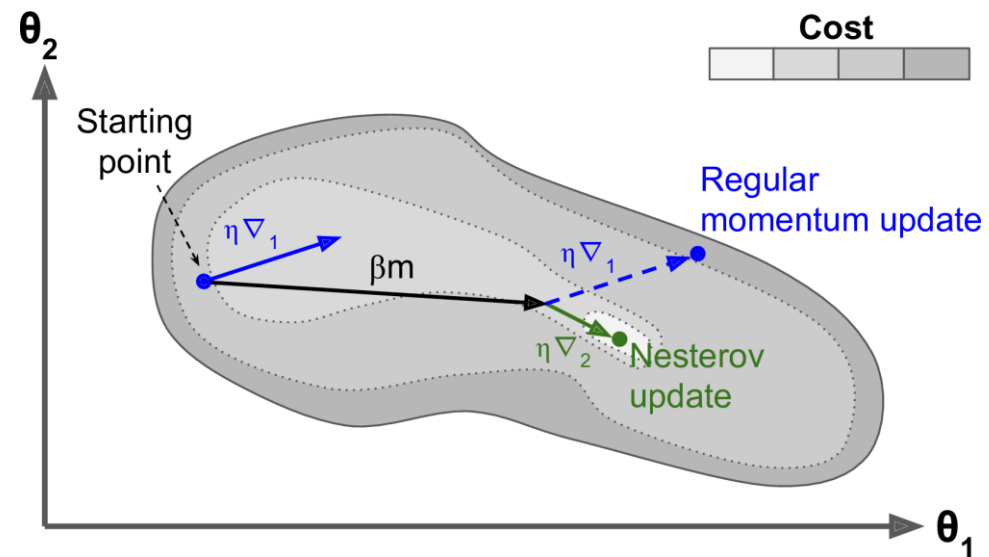
β

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

4. Faster Optimizers

- **Nesterov momentum optimization** measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta\mathbf{m}$

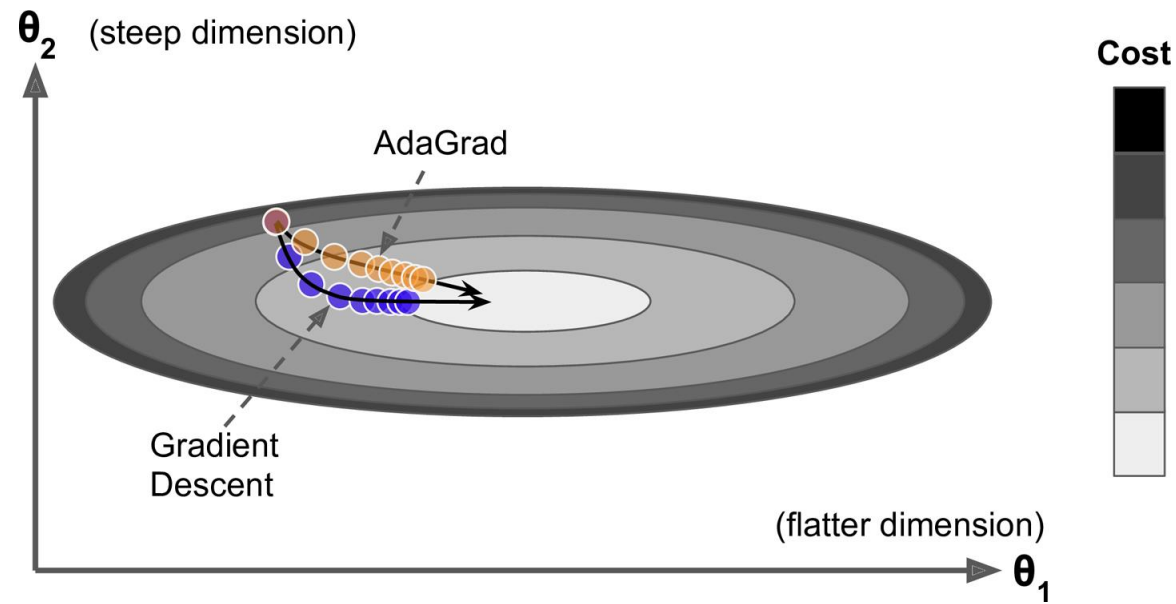
1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$



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


4. Faster Optimizers

- The **adaptive optimizers** such as **AdaGrad**, **RMSProp**, **Adam**, and **Nadam** scale down the gradient vector along the steepest dimensions.



```
optimizer = keras.optimizers.RMSprop()  
optimizer = keras.optimizers.Adam()
```

4. Faster Optimizers

- RMSProp, Adam and Nadam often **converge fast**. But they can give poor **generalization**.
- Solution: Use Nesterov accelerated gradient.

Class	Speed	Quality
SGD	*	***
SGD with momentum, Nesterov	**	***
Adagrad	***	*
RMSProp, Adam, Nadam, AdaMax	***	** or ***

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

5. Avoiding Overfitting

- Deep neural networks typically have many parameters, giving them **ability to fit** a huge variety of complex datasets.
- **Useful regularization techniques:**
 - Early stopping
 - Batch normalization
 - ℓ_1 and ℓ_2 regularization
 - Dropout

5.1 ℓ_1 and ℓ_2 Regularization

- Constrain a neural network's connection weights.

- ℓ_1 :
$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum \|w\|$$

- ℓ_2 :
$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m} * \sum \|w\|^2$$

```
layer = keras.layers.Dense(100, activation="elu",  
                             kernel_initializer="he_normal",  
                             kernel_regularizer=keras.regularizers.l1(0.01))
```

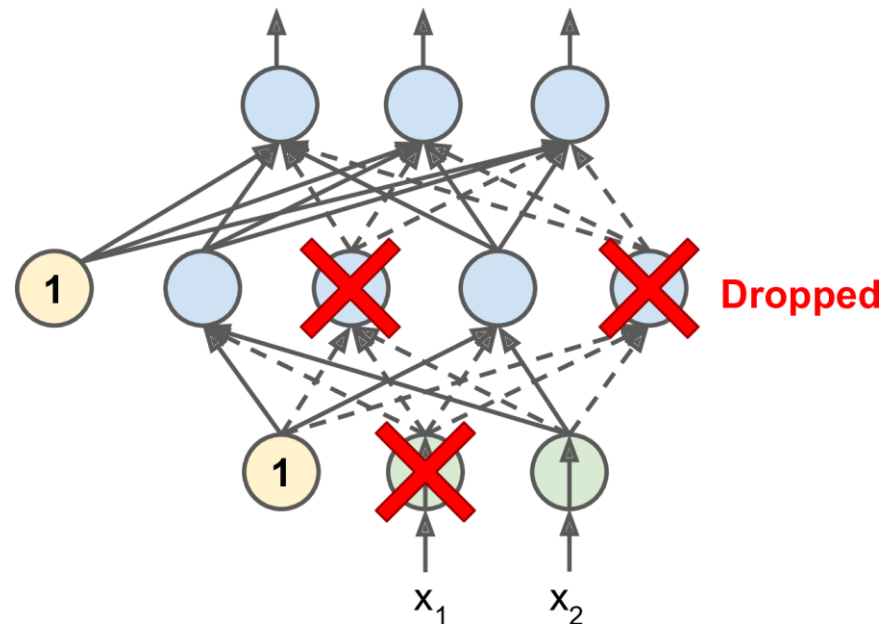
The other regularization functions:

```
keras.regularizers.l2(0.01)
```

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

5.2 Dropout

- Popular technique to improve accuracy.
- At every training step, every neuron (excluding the output neurons) has a probability p of being temporarily **dropped out**.



5.2 Dropout

```
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")
])
```

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
 - Glorot and He Initialization
 - Nonsaturating Activation Functions
 - Batch Normalization
 - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Avoiding Overfitting
 - ℓ_1 and ℓ_2 Regularization
 - Dropout
6. Summary
7. Exercise

6. Summary

- **Recommended default DNN** configuration

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	1 cycle

6. Summary

- For a simple **stack of dense** or **CNN layers**.

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1 cycle

7. Exercise

11.8. Practice training a deep neural network on the **CIFAR10 image dataset**:

- a) Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the ELU activation function.
- b) Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with `keras.datasets.cifar10.load_data()`. The dataset is composed of 60,000 32 × 32-pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you'll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model's architecture or hyperparameters.
- c) Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?
- d) Try replacing Batch Normalization with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
- e) Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.
- f) Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.