Co-funded by the
Erasmus+ Programme
of the European Union

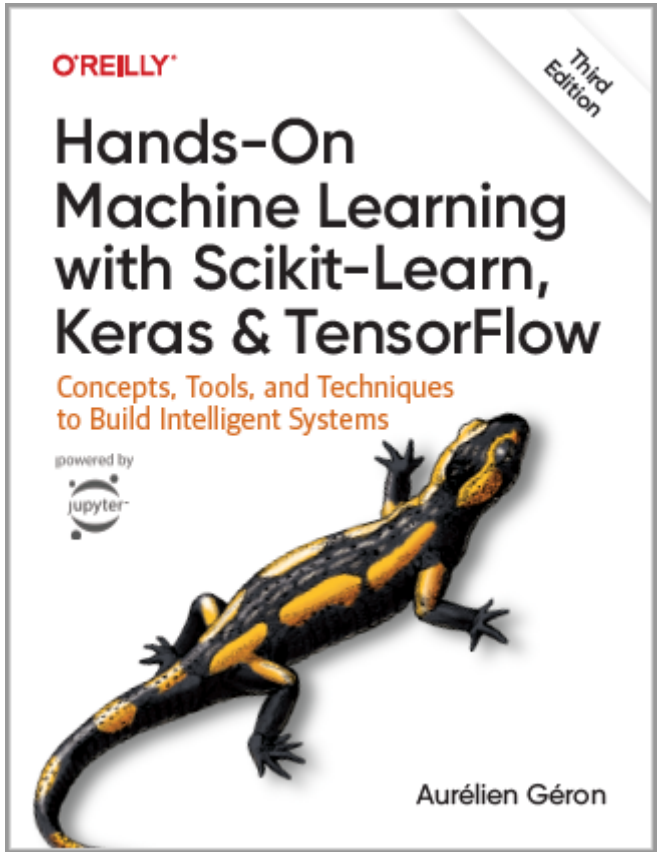
DeCAIR

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, 3rd Edition, 2022
  - Material: https://github.com/ageron/handson-ml3

Outline

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

1. Introduction

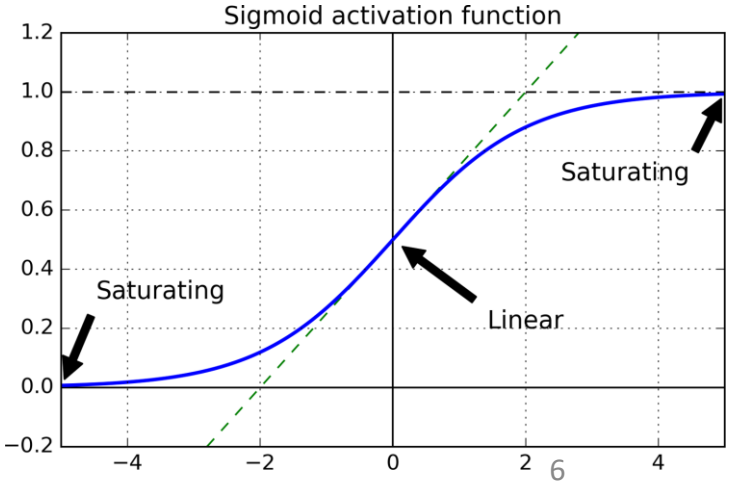
- Deep neural networks can solve **complex problems** and provide **end-to-end** solutions.
- When you train a deep network, you may face the following **problems**:
  - **Vanishing** or **exploding gradients**: The gradients grow smaller and smaller, or larger and larger.
  - **Not enough data**
  -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

Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
   - Glorot and He Initialization
   - Better Activation Functions
   - Batch Normalization
   - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Learning Rate Scheduling
6. Avoiding Overfitting
   - ℓ_1 and ℓ_2 Regularization
   - Dropout
7. Summary
8. 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' connection 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**: 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 $\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$

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

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

2.1 Glorot and He Initialization

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

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, sigmoid, softmax	$1 / fan_{avg}$
He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / fan_{in}$
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**:

```
layers.Dense(50, activation="relu",
     kernel_initializer="he_normal") # Or "he_uniform"
```

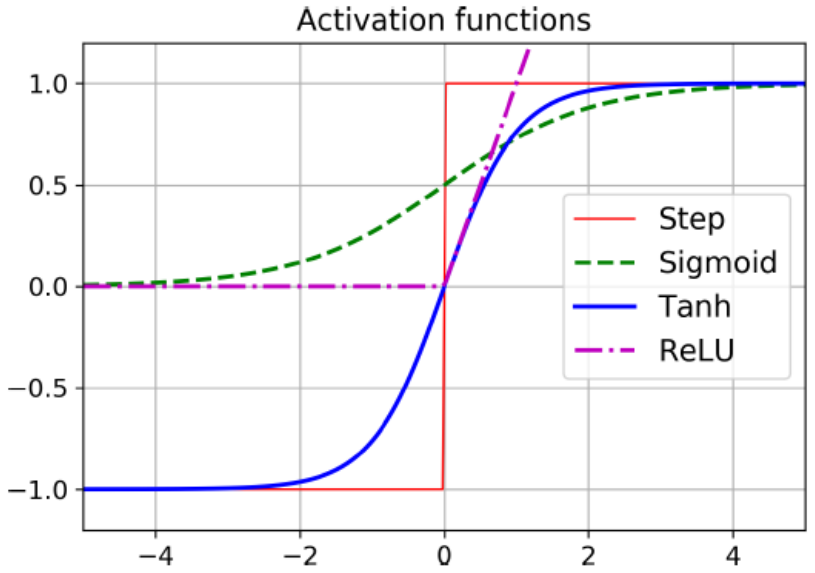
- **He initialization** with a **uniform** distribution but based on $\mathbf{fan_{avg}}$:

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

keras.layers.Dense(50, activation="sigmoid",
                   kernel_initializer=he_avg_init)
```

2.2 Better 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 problem**: A neuron dies when its input is negative for all training instances.

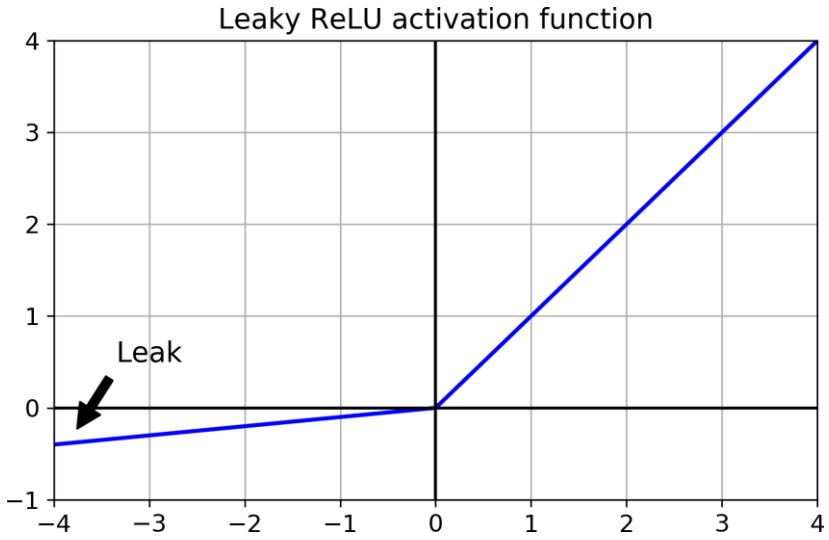


2.2 Better Activation Functions

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

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

- **α** between 0.01 and 0.3

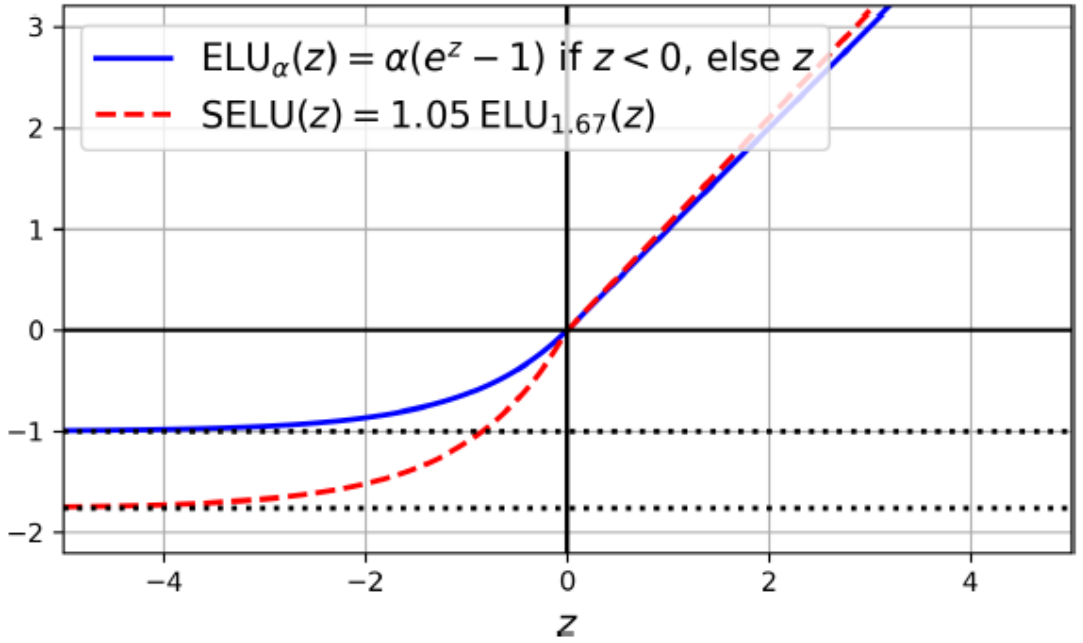



```
model = keras.models.Sequential([
    …
    layers.Dense(50, kernel_initializer="he_normal"),
    layers.LeakyReLU(alpha=0.2), # added as a layer
    …
])
```

2.2 Better Activation Functions

- **Exponential linear unit (ELU)** also performs better than ReLU but is slower.
- **Scaled ELU (SELU)** performs best with MLP networks.
- **Self-normalize networks**: Scale inputs, SELU, and **`lecun_normal`**, no other regularization.

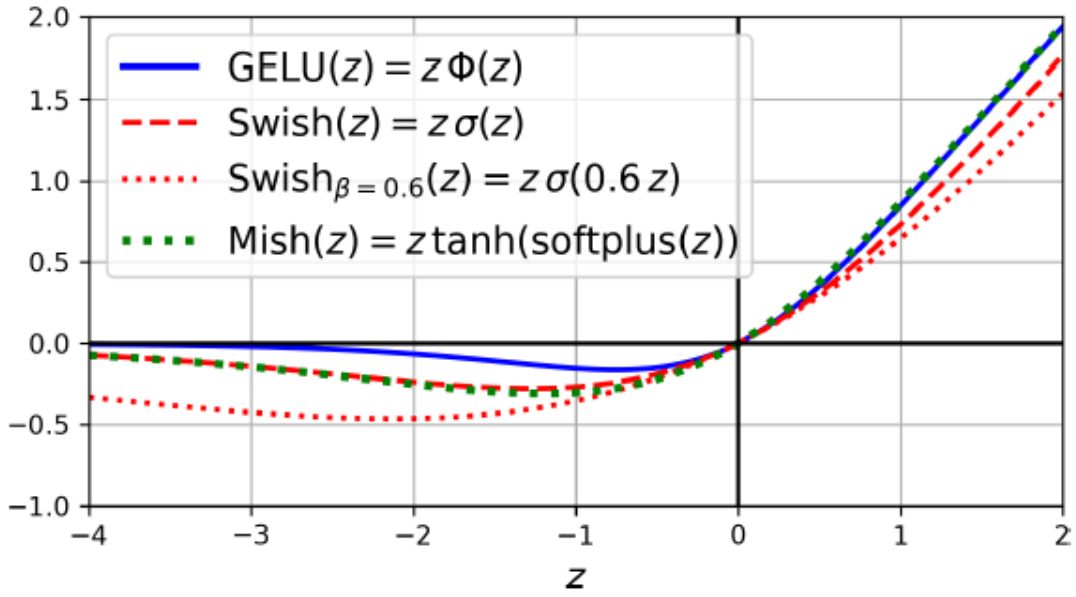
$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

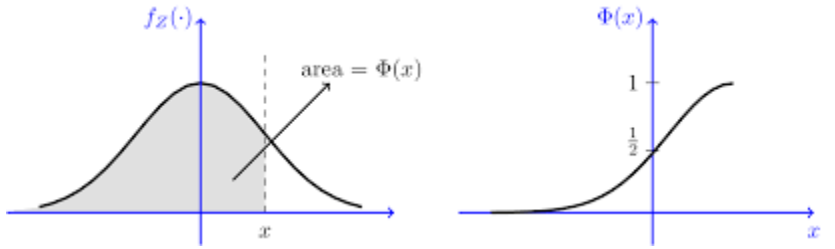



```
layer = layers.Dense(10, activation="selu",
    kernel_initializer="lecun_normal")
```

2.2 Better Activation Functions

- **GELU:** $z\Phi(z)$, where $\Phi(z)$ is the Gaussian CDF.
- **Swish:** Can be parametrized $\text{Swish}_\beta(z) = z\sigma(\beta z)$.
- **Mish:** $z\ \tanh(\text{softplus}(z)))$, where $\text{softplus}(z) = \log(1 + \exp(z))$.





2.2 Better Activation Functions

- **Summary:**
  - **Results**: Mish > Swish > GELU > SELU > ELU > leaky ReLU > ReLU > tanh > logistic
  - **Speed**: ReLU > leaky ReLU > ELU > SELU > Swish > Mish > GELU
- For deep MLP, try SELU.
- For simple tasks or fast response, use ReLU.
- For complex tasks and fast response, use leaky ReLU.

- **Names in Keras**
  - `elu`
  - `gelu`
  - `linear`
  - `relu`
  - `selu`
  - `sigmoid`
  - `softmax`
  - `swish`
  - `tanh`

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**.

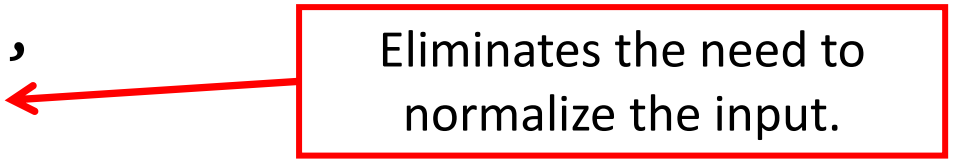
- **Batch Normalization (BN)** zero-centers and normalizes each layer input using statistics from the mini batch (> 30).

- **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.Sequential([
        layers.Flatten(input_shape=[28, 28]),
        layers.BatchNormalization(),
        layers.Dense(300, activation="relu",
                     kernel_initializer="he_normal"),
        layers.BatchNormalization(),
        layers.Dense(100, activation="relu",
                     kernel_initializer="he_normal"),
        layers.BatchNormalization(),
        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.

- To clip the gradient vector to a value between -1.0 and 1.0:

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

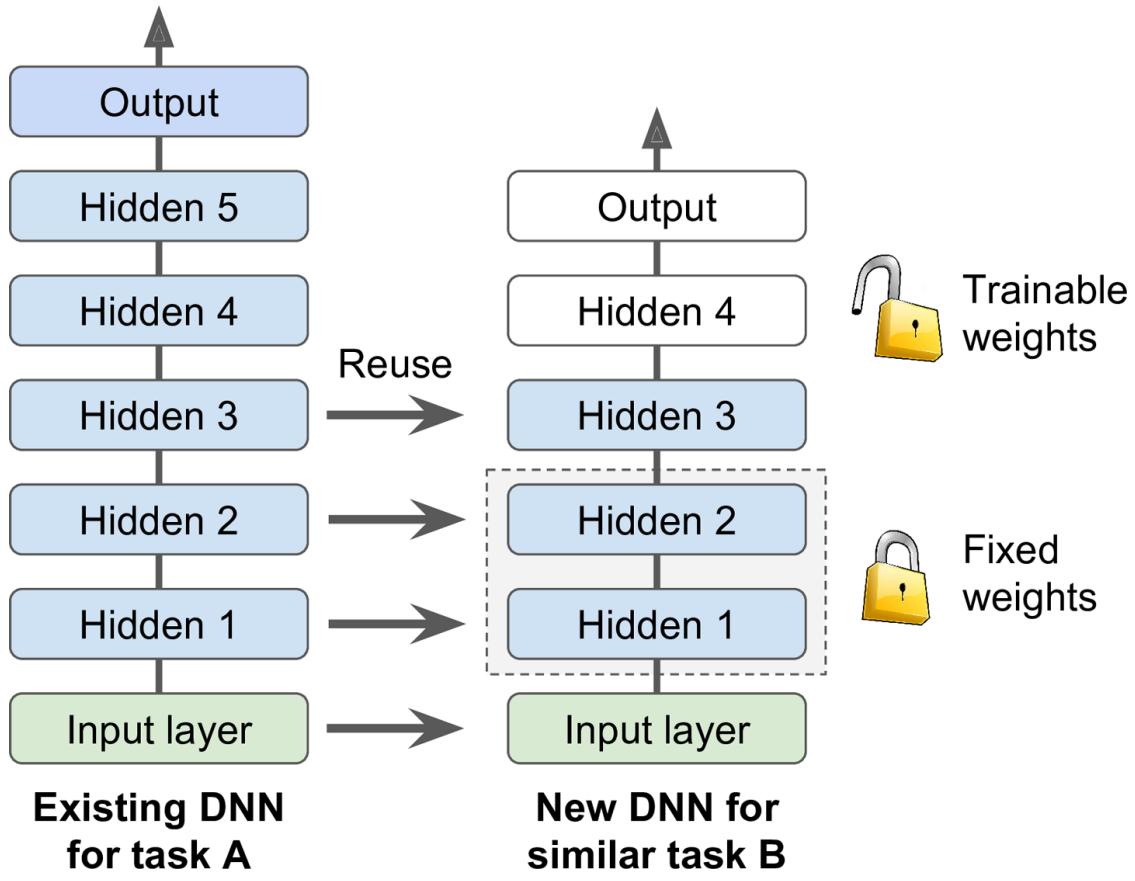
Outline

1. Introduction
2. Vanishing/Exploding Gradients Problems
   - Glorot and He Initialization
   - Better Activation Functions
   - Batch Normalization
   - Gradient Clipping
3. Reusing Pretrained Layers
4. Faster Optimizers
5. Learning Rate Scheduling
6. Avoiding Overfitting
   - ℓ_1 and ℓ_2 Regularization
   - Dropout
7. Summary
8. 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, e.g., classifies 8 classes
model_A = keras.models.load_model("my_model_A")
# Create a new model (binary classifier) using all but the last layer
model_B_on_A = keras.Sequential(model_A.layers[:-1])
model_B_on_A.add(layers.Dense(1, activation="sigmoid"))
# Freeze loaded layers then compile
for layer in model_B_on_A.layers[:-1]:
        layer.trainable = False

optimizer = keras.optimizers.SGD(learning_rate=0.001)
model_B_on_A.compile(loss="binary_crossentropy",
        optimizer=optimizer, 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))
# Unreeze loaded layers
for layer in model_B_on_A.layers[:-1]:
        layer.trainable = True
# Compile with small learning rate (defalut = 1e-2)
optimizer = keras.optimizers.SGD(learning_rate=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))
```

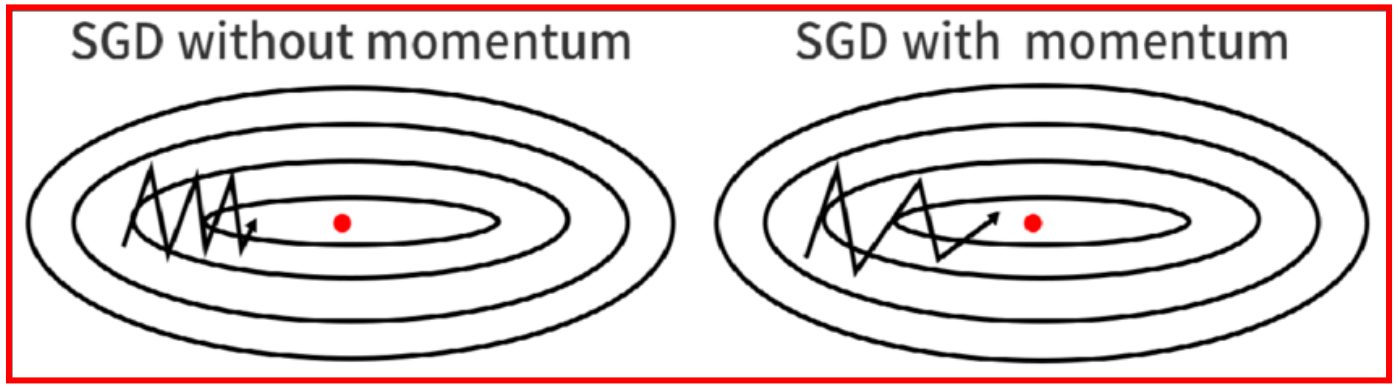
Test accuracy without transfer learning = 91.85%
Test accuracy with transfer learning = 93.85%

Outline

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

4. Faster Optimizers

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



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

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

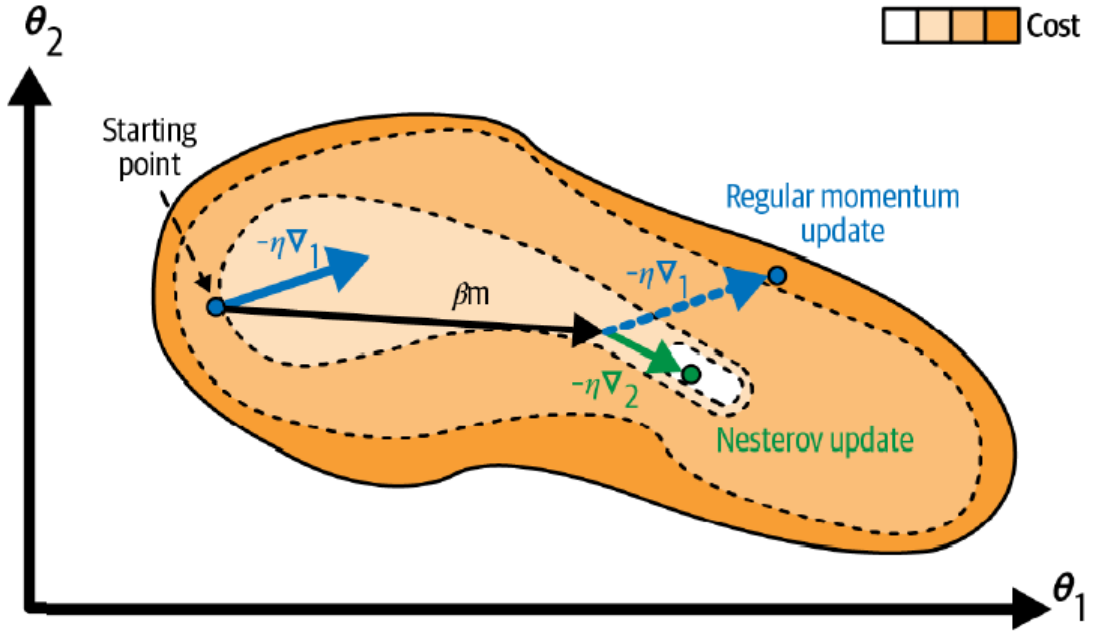
β

```
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 **θ** + β**m**

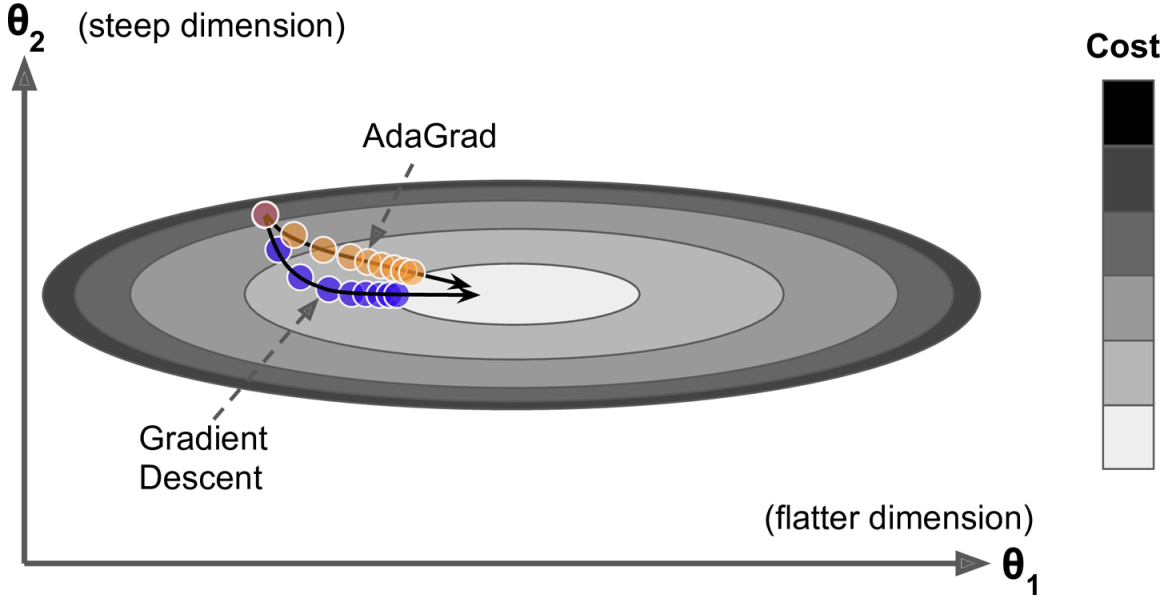
1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\boldsymbol{\theta} + \beta\mathbf{m})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$




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

4. Faster Optimizers

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




```
optimizer = keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)
optimizer = keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
```

4. Faster Optimizers

- The adaptive optimizers 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, AdaMax, Nadam, AdamW	***	** or ***

Outline

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

5. Learning Rate Scheduling

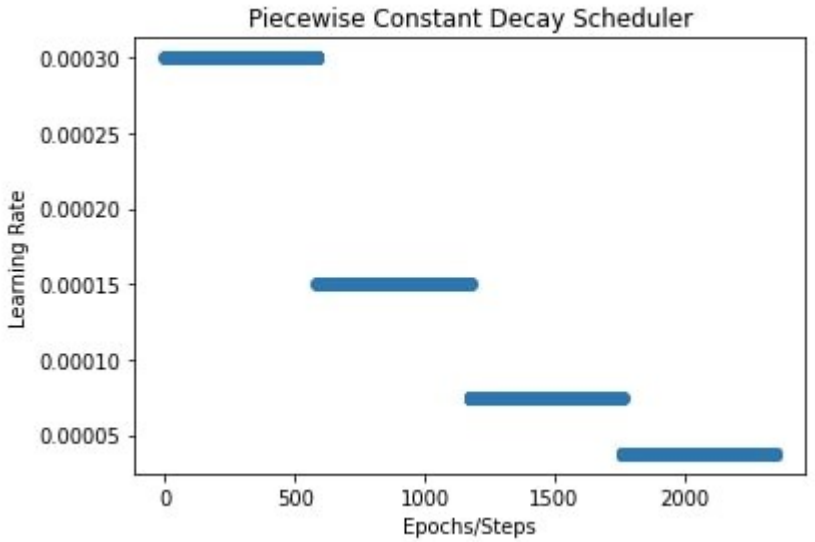
- The learning rate affects the **learning speed** and **model quality**.
- **LR Scheduling**: Best to start with a large learning rate and then reduce it.

5. LR Scheduling Strategies

1. **Power scheduling** (Easy) $\eta(t) = \eta_0 \,/\, (1 + t/s)^c$
   - η_0: initial rate, t: time in steps, s: number of steps, c: usually 1

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.01, decay=1e-4)
```

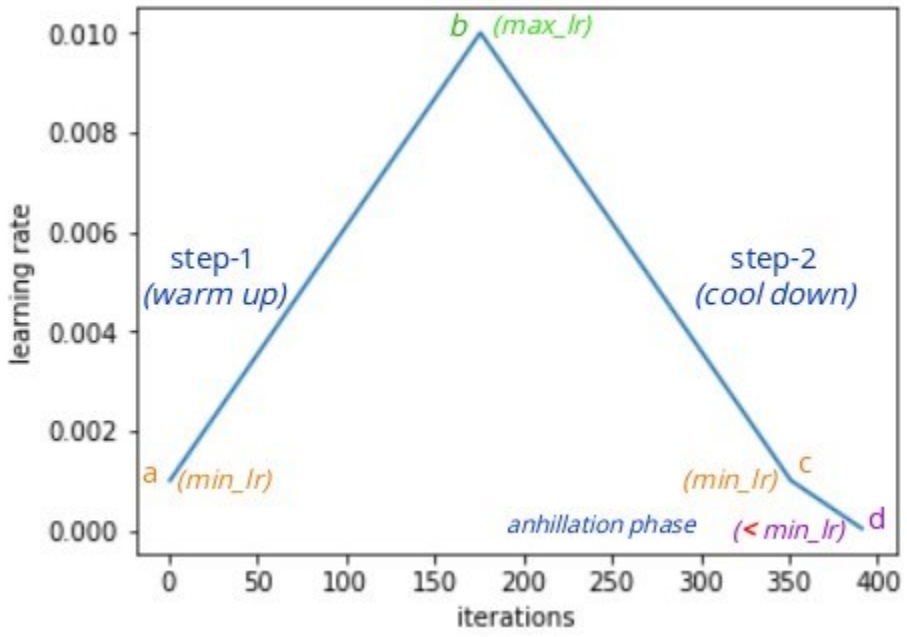
   - decay = $1/s$
2. **Exponential scheduling** (Good) $\eta(t) = \eta_0\, 0.1^{t/s}$
3. **Piecewise constant scheduling** (Difficult)



5. LR Scheduling Strategies

4. **Performance Scheduling** (Good): reduce the learning rate by a factor of λ when the validation error stops dropping.

5. **One-cycle scheduling** (Excellent)



Outline

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

6. 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

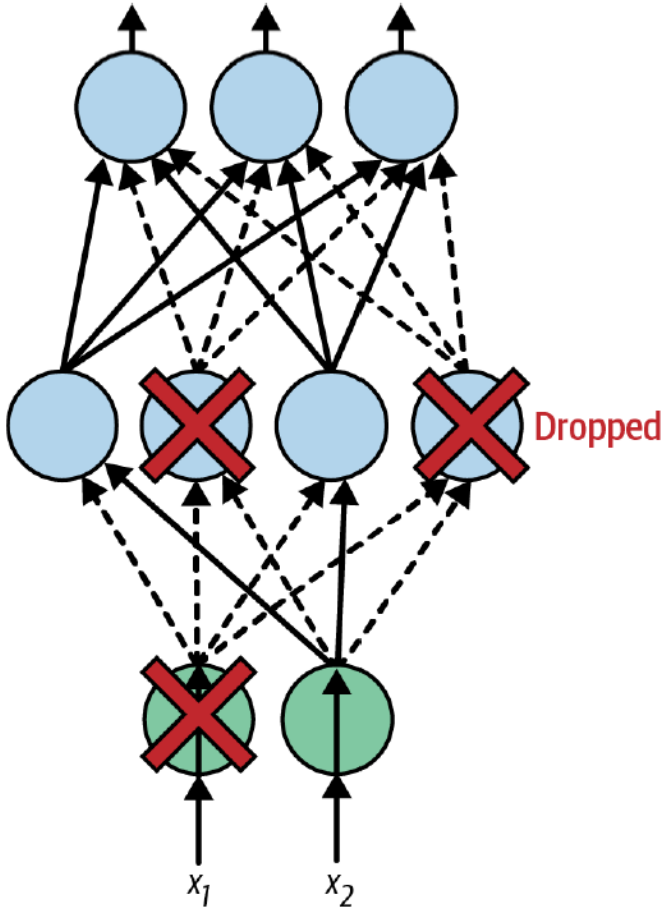
6.1 ℓ_1 and ℓ_2 Regularization

- Constrain a neural network's connection weights.
  - ℓ_1: $J(\mathbf{\theta}) = \mathrm{MSE}(\mathbf{\theta}) + \frac{\alpha}{m}\Sigma_{i=1}^{n}\theta_i^2$
  - ℓ_2: $J(\mathbf{\theta}) = \mathrm{MSE}(\mathbf{\theta}) + 2\alpha\Sigma_{i=1}^{n}|\theta_i|$

```
layer = layers.Dense(100, activation="relu",
       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)
```

6.2 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**.



6.2 Dropout

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

Outline

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

7. Summary

- **Recommended default DNN** configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1 cycle

7. Summary

- For a simple **stack of dense** or **CNN layers** (self-normalizing net).

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1 cycle

8. 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 **Swish** 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.

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**.