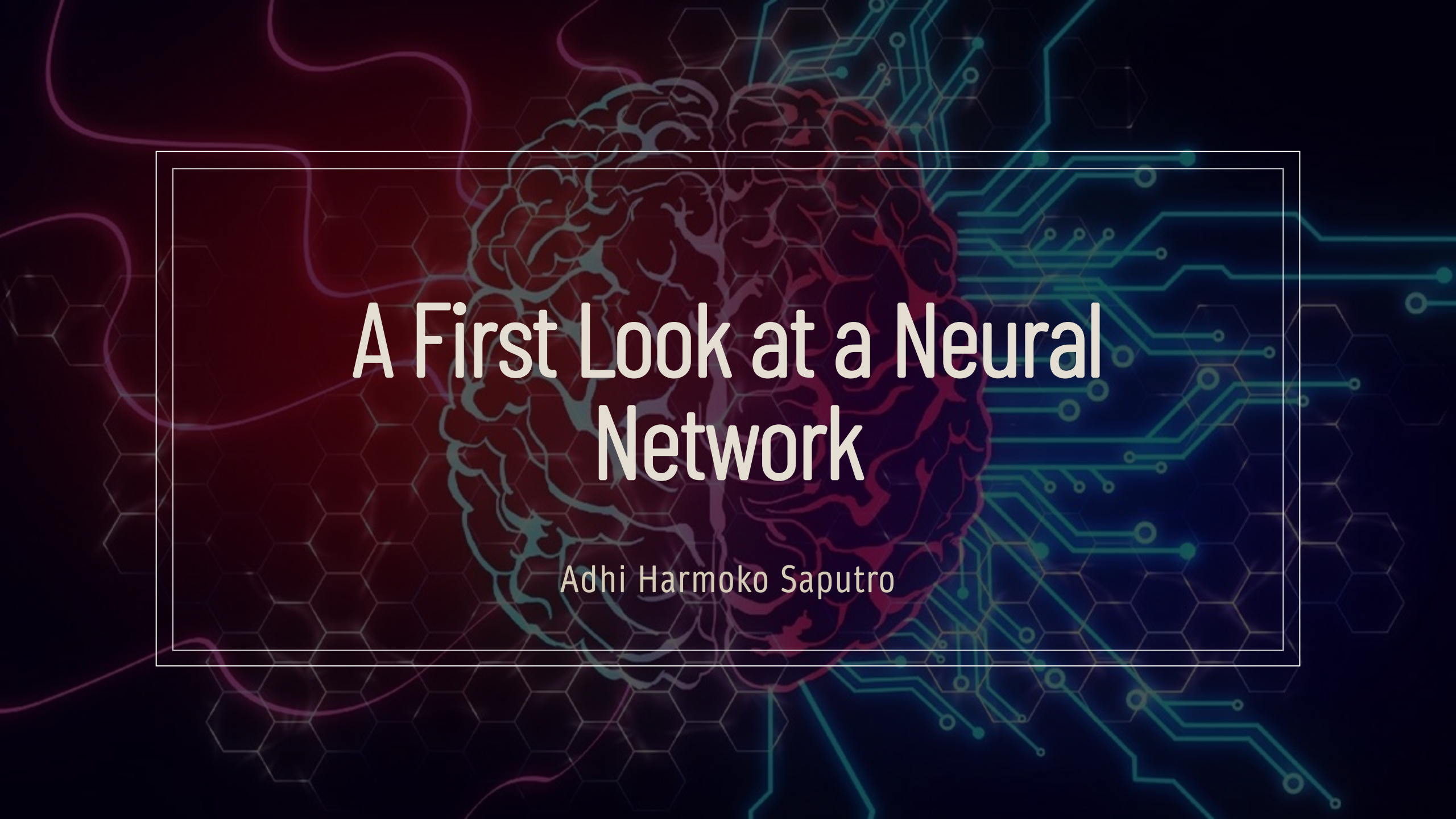
Artificial Intelligence

Adhi Harmoko Saputro

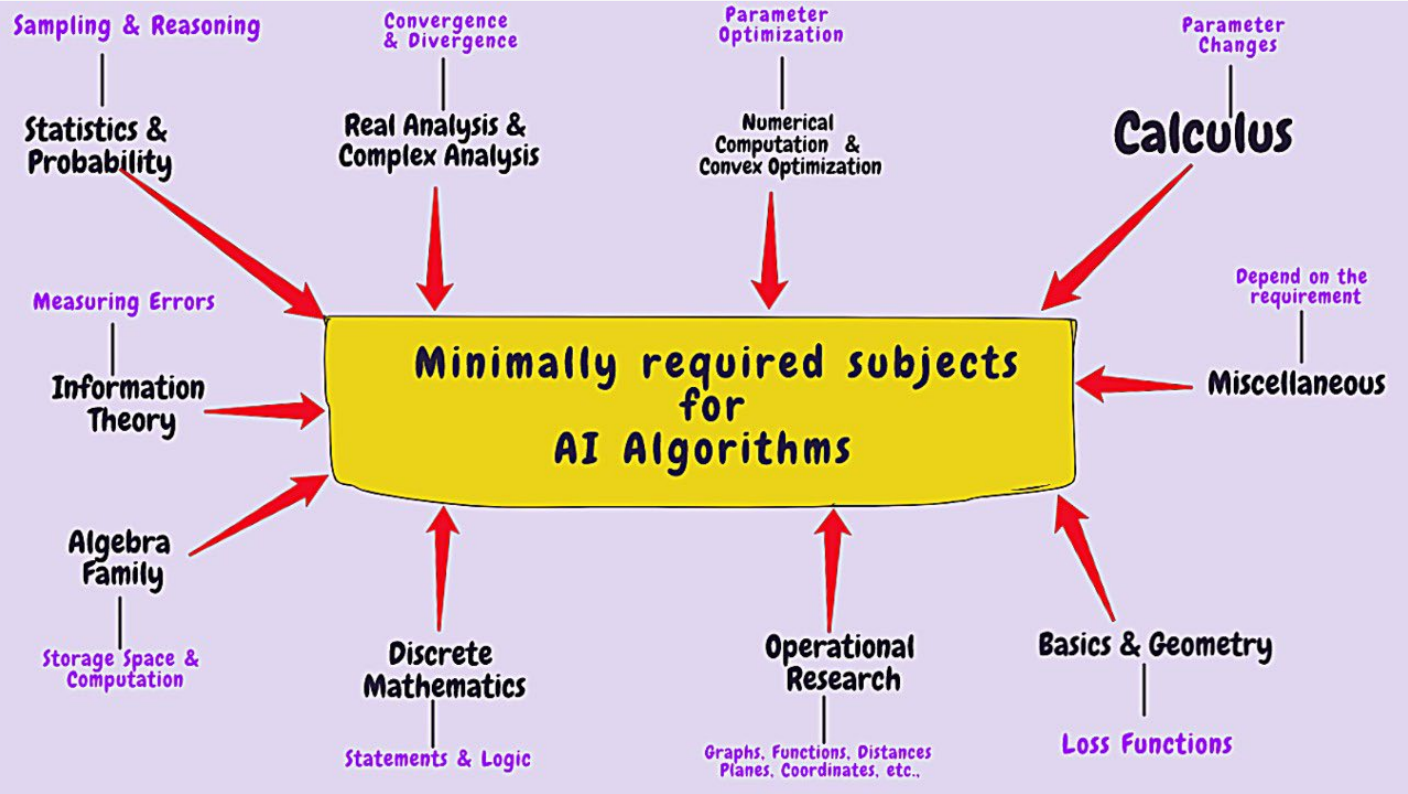


A First Look at a Neural Network

Adhi Harmoko Saputro

Mathematical Concepts Needs

- Understanding deep learning requires familiarity with many simple mathematical concepts: tensors, tensor operations, differentiation, gradient descent, and so on



A first look at a Neural Network

- A concrete example of a neural network that uses the Python library Keras to learn to classify handwritten digits
  - Classify grayscale images of handwritten digits (28 × 28 pixels) into their 10 categories (0 through 9)

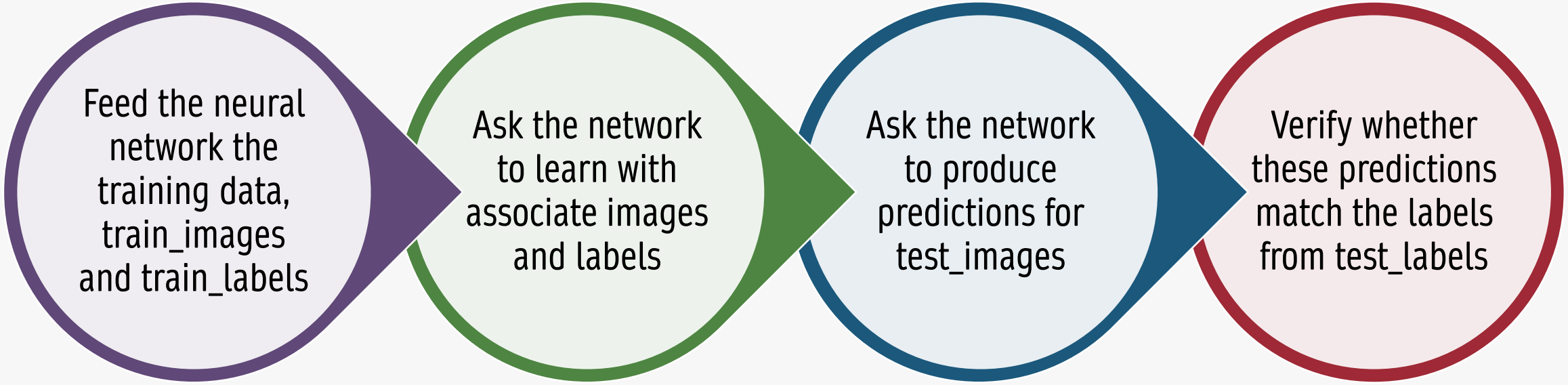
  - In Deep Learning:
    - A category in a classification problem is called a class
    - Data points are called samples
    - The class associated with a specific sample is called a label

Loading the MNIST Dataset in Keras

- Assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s
  - 60,000 training images
  - 10,000 test images
  - The images and labels have a one-to-one correspondence
- The training data:
  - train_images and train_labels form the training set
  - the data that the model will learn from
- The test data:
  - The model will then be tested on the test set, test_images and test_labels

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()



train_images.shape
(60000, 28, 28)

len(train_labels)
60000

train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)

test_images.shape
(10000, 28, 28)

len(test_labels)
10000

test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Simple Development Workflow



The Layer

- The core building block of neural networks is the layer
  - A layer is a filter for data:
    - Some data goes in
    - Comes out in a more useful form
  - Layers extract representations out of the data fed into them
    - Representations that are more meaningful for the problem at hand
  - Most of deep learning consists of chaining together simple layers
    - Implement a form of progressive data distillation

- A deep learning model is like a sieve for data processing
  - Made of a succession of increasingly refined data filters as the layers

The Network Architecture

- The model consists of a sequence of two Dense layers
  - Densely connected (also called fully connected) neural layers
- The second (and last) layer is a 10-way softmax classification layer
  - It return an array of 10 probability scores (summing to 1)
- Each score will be the probability that the current digit image belongs to one of our 10 digit classes

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

The Compilation Step

- An optimizer
  - The mechanism through which the model will update itself based on the training data it sees, so as to improve its performance
- A loss function
  - How the model will be able to measure its performance on the training data
  - How it will be able to steer itself in the right direction
- Metrics to monitor during training and testing
  - Care about accuracy (the fraction of the images that were correctly classified)

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Preparing The Image Data

- Preprocess the data by reshaping it into the shape the model expects
- Scaling it so that all values are in the [0, 1] interval
- Training images were stored in an array of shape (60000, 28, 28) of type uint8 with values in the [0, 255] interval
- Transform it into a float32 array of shape (60000, 28 *28) with values between 0 and 1

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

print(train_images.shape)
print(test_images.shape)

(60000, 784)
(10000, 784)
```

Train The Model

- Train the model, which in Keras is done via a call to the model's fit() method
  - Fit the model to its training data
- Two quantities are displayed during training:
  - The loss of the model over the training data
  - The accuracy of the model over the training data
- Quickly reach an accuracy of 0.9833 (98.33%) on the training data

```
model.fit(train_images, train_labels, epochs=5,
batch_size=128)

Epoch 1/50 469/469 [==============================] - 1s 3ms/step - loss:
0.3037 - accuracy: 0.9130
Epoch 2/50 469/469 [==============================] - 1s 2ms/step -
loss: 0.2888 - accuracy: 0.9170
Epoch 3/50 469/469 [==============================] - 1s 2ms/step -
loss: 0.2759 - accuracy: 0.9202
Epoch 49/50 469/469 [==============================] - 1s 2ms/step -
loss: 0.0608 - accuracy: 0.9831
Epoch 50/50 469/469 [==============================] - 1s 3ms/step -
loss: 0.0595 - accuracy: 0.9833
<tensorflow.python.keras.callbacks.History at 0x1e50ac54e80>
```

Using The Model to Make Predictions

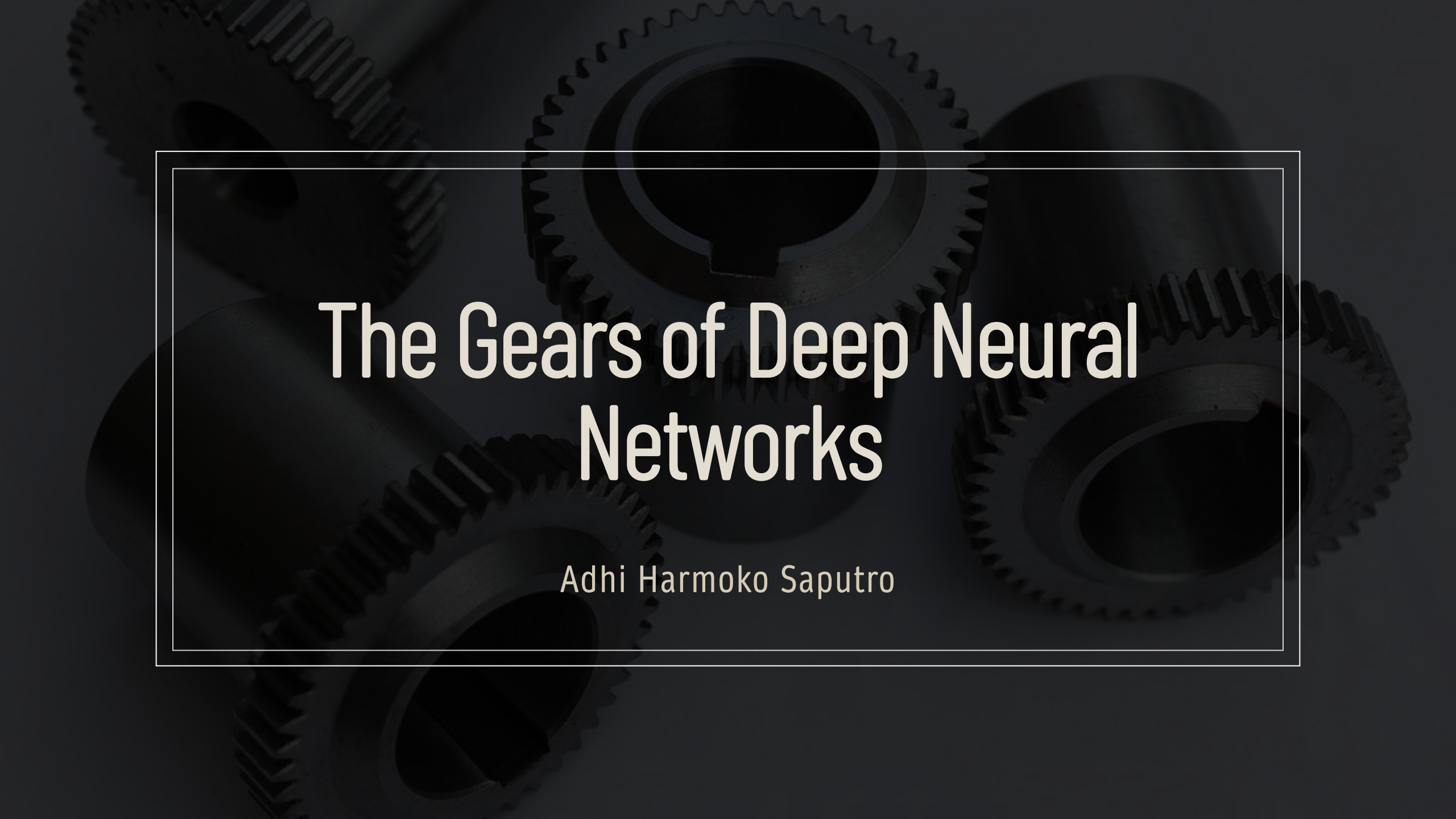
- Have a trained model
  - Can use it to predict class probabilities for new digits
  - Images that weren't part of the training data, like those from the test set
- Each number of index i in that array corresponds to the probability that digit image test_digits[0] belongs to class i
- This first test digit has the highest probability score (0.99853909, almost 1) at index 7
  - It must be a 7
- Can check that the test label agrees

```
test_digits = test_images[0:10]
predictions = model.predict(test_digits)
predictions[0]
array([2.3584458e-07,
       3.7251192e-11,
       5.0270013e-05,
       1.3622823e-03,
       7.4093059e-10,
       2.4309165e-06,
       4.2852251e-14,
       9.9853909e-01,
       5.3168451e-06,
       4.0446674e-05], dtype=float32)

predictions[0].argmax()
7

predictions[0][7]
0.9985391

test_labels[0]
7
```

Evaluating the Model on New Data

- How good is the model at classifying such never-before-seen digits?
  - Check by computing average accuracy over the entire test set
- The test-set accuracy turns out to be 97.4%
  - Quite a bit lower than the training set accuracy 98.33%
- This gap between training accuracy and test accuracy is an example of overfitting:
  - The fact that machine learning models tend to perform worse on new data than on their training data
  - Overfitting is a central topic in the next course

```
test_loss, test_acc = model.evaluate(test_images,
test_labels)
print(f"test_acc: {test_acc}")

313/313 [==============================] - 1s 2ms/step -
loss: 0.0863 - accuracy: 0.9741
test_acc: 0.9740999937057495
```



The Gears of Deep Neural Networks

Adhi Harmoko Saputro

Tensor Operations

- All transformations learned by deep neural networks can be reduced to a handful of tensor operations (or tensor functions) applied to tensors of numeric data
  - Possible to add tensors, multiply tensors, and so on
- Example: Built a model by stacking Dense layers on top of each other using Keras
- This layer can be interpreted as a function, which takes as input a matrix and returns another matrix
  - The function is relu
    - W is a matrix
    - b is a vector

```
keras.layers.Dense(512, activation="relu")
```

```
output = relu(dot(input, W) + b)
```

A relu operation: relu(x) is max(x, 0); “relu” stands for “rectified linear unit”

An addition (+) between the resulting matrix and a vector b

A dot product (dot) between the input tensor and a tensor named W

Element-wise Operations

- The relu operation and addition are element-wise operations:
  - Operations that are applied independently to each entry in the tensors
  - Write a naive Python implementation of an element-wise operation
    - Use a for loop, as in this naive implementation of an element-wise relu operation

```
def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

x is a rank-2 NumPy tensor

Avoid overwriting the input tensor

- Could do the same for addition

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

x and y are rank-2 NumPy tensors

- On the same principle, can do element-wise multiplication, subtraction, and so on

Element-wise Operations

- In practice, when dealing with NumPy arrays
  - These operations are available as well optimized built-in NumPy functions, which themselves delegate the heavy lifting to a Basic Linear Algebra Subprograms (BLAS) implementation

- In NumPy, can do the following element-wise operation, and it will be blazing fast

```
import numpy as np

z = x + y
z = np.maximum(z, 0.)
```

Element-wise Operations

- Actually time the difference between naive Python and NumPy element-wise operation

- When running TensorFlow code on a GPU, element-wise operations are executed via fully vectorized CUDA implementations that can best utilize the highly parallel GPU chip architecture

```
import numpy as np
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {0:.5f} s".format(time.time() - t0))
Took: 0.00499 s


t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.5f} s".format(time.time() - t0))
Took: 1.86103 s
```

Broadcasting

- The smaller tensor will be broadcast to match the shape of the larger tensor (If possible and no ambiguity)
  - Axes (called broadcast axes) are added to the smaller tensor to match the ndim of the larger tensor
  - The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor
- Consider X with shape (32, 10) and y with shape (10,)
- Add an empty first axis to y, whose shape becomes (1, 10)
- Repeat y 32 times alongside this new axis
  - End up with a tensor Y with shape (32, 10), where Y[i, :] == y for i in range(0, 32):
  - Can proceed to add X and Y since have the same shape

```
import numpy as np
X = np.random.random((32, 10))
y = np.random.random((10,))

y = np.expand_dims(y, axis=0)


Y = np.concatenate([y] * 32, axis=0)
```

X is a random matrix with shape (32, 10)

y is a random vector with shape (10,)

The shape of y is now (1, 10)

Repeat y 32 times along axis 0 to obtain Y, which has shape (32, 10)

Broadcasting

- The repetition operation is entirely virtual
  - It happens at the algorithmic level rather than at the memory level.
  - But thinking of the vector being repeated 10 times alongside a new axis is a helpful mental model
  - Here's what a naïve implementation would look like
  - With broadcasting, can generally perform element-wise operations that take two inputs tensors if one tensor has shape (a, b, ... n, n + 1, ... m) and the other has shape (n, n + 1, ... m)
  - The broadcasting will then automatically happen for axes a through n – 1

- Example: applies the element-wise maximum operation to two tensors of different shapes via broadcasting

The output z has shape (64, 3, 32, 10) like x

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a rank-2 NumPy tensor

y is a NumPy vector

Avoid overwriting the input tensor

```
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10)

y is a random tensor with shape (32, 10)

Tensor Product

- In NumPy, a tensor product is done using the np.dot function
  - The mathematical notation for tensor product is usually a dot
- The dot product of two vectors, x and y
- Remember
  - The dot product between two vectors is a scalar
  - Only vectors with the same number of elements are compatible for a dot product

```
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)
```

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are NumPy vectors

Tensor Product

- Take the dot product between a matrix x and a vector y, which returns a vector where the coefficients are the dot products between y and the rows of x

- Take the dot product of two matrices x and y
  - (dot(x, y)) if and only if x.shape[1] == y.shape[0]

```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z


def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

x is a NumPy matrix

y is a NumPy vector

This operation returns a vector of 0s with the same shape as y

The first dimension of x must be the same as the 0th dimension of y

Tensor Reshaping

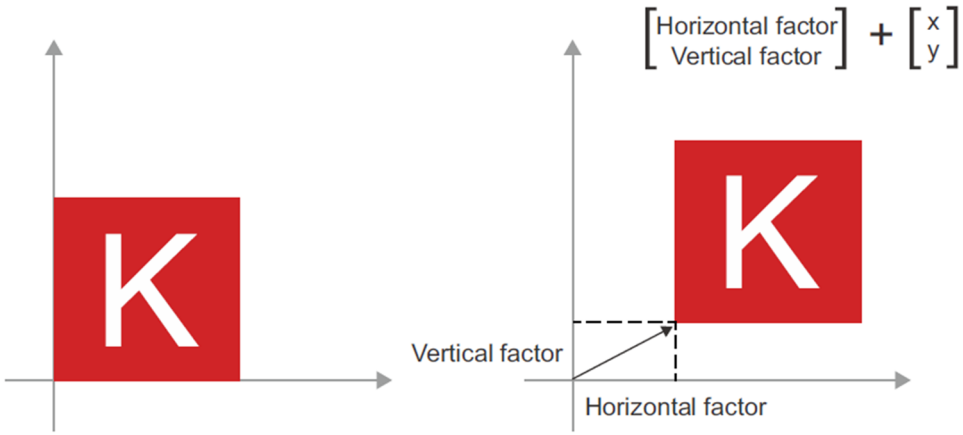
- Reshaping a tensor means rearranging its rows and columns to match a target shape
  - The reshaped tensor has the same total number of coefficients as the initial tensor

```
x = np.array([[0., 1.],
              [2., 3.],
              [4., 5.]])
x.shape
(3, 2)


x = x.reshape((6, 1))
x
array([[0.], [1.], [2.], [3.], [4.], [5.]])


x = x.reshape((2, 3))
x
array([[0., 1., 2.], [3., 4., 5.]])


x = np.zeros((300, 20))
x = np.transpose(x)
x.shape
(20, 300)
```

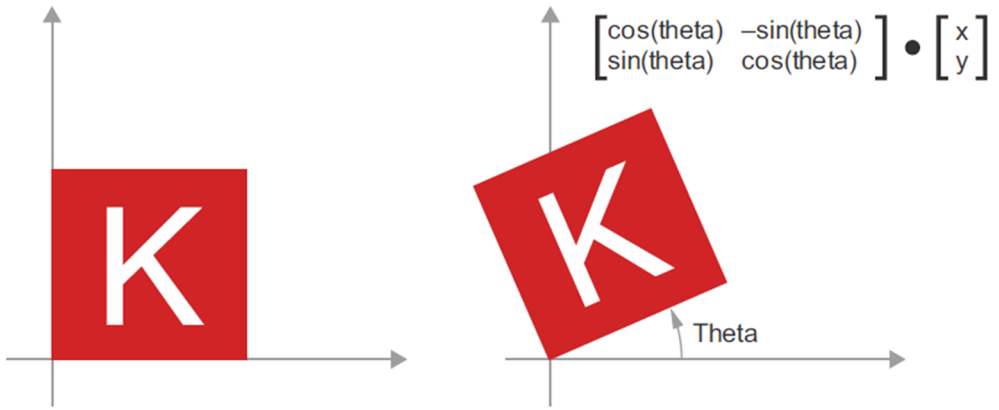
- Transposing a matrix means exchanging its rows and its columns, so that x[i, :] becomes x[:, i]:

Geometric Interpretation of Tensor Operations

- Translation:
  - Moving the object without distorting it, by a certain amount in a certain direction
  - Adding a vector to a point will move the point by a fixed amount in a fixed direction
  - Applied to a set of points (such as a 2D object)

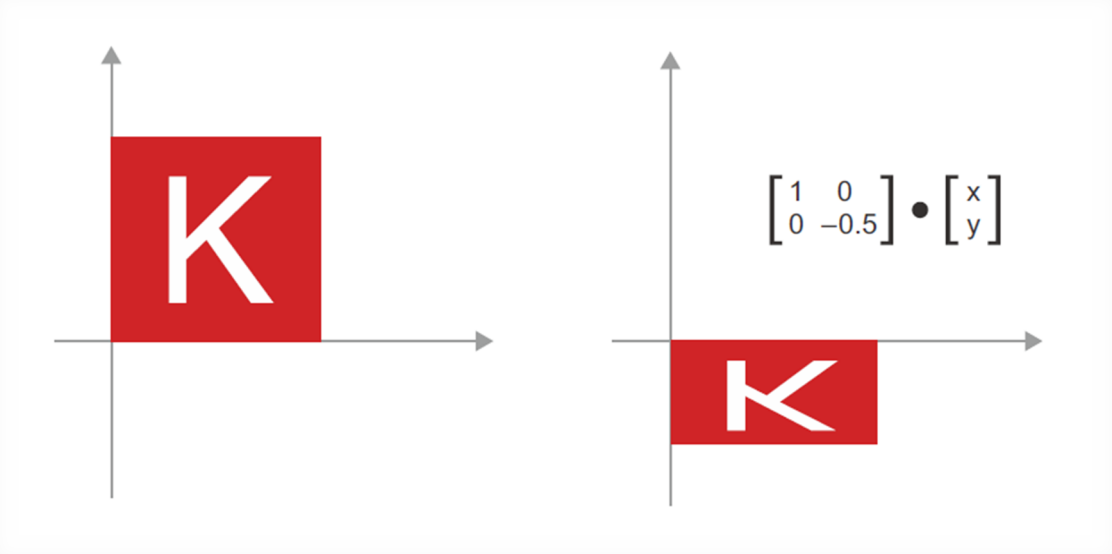


- Rotation:
  - A counterclockwise rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2 × 2 matrix
  - R = [[cos(theta), -sin(theta)], [sin(theta), cos(theta)]]



Geometric Interpretation of Tensor Operations

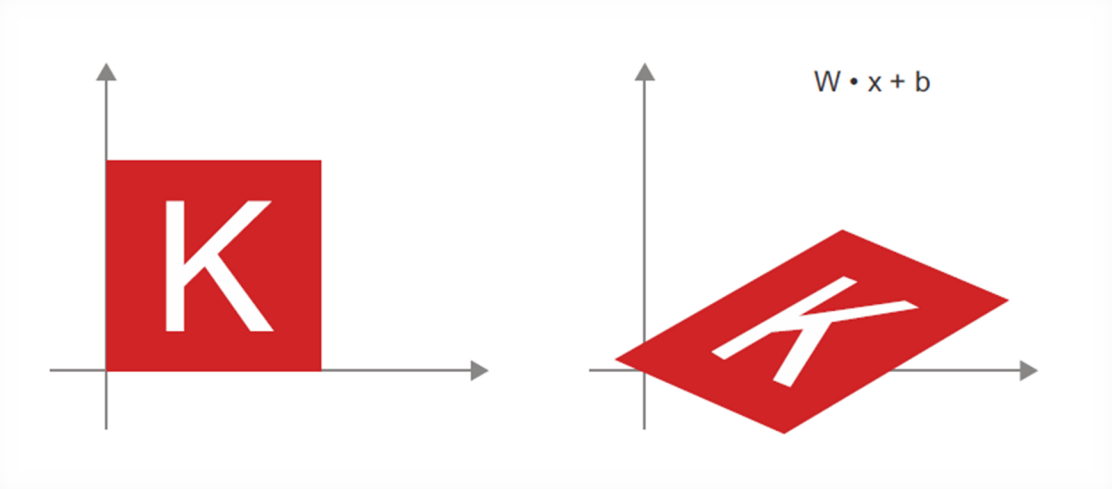
- Scaling:
  - A vertical and horizontal scaling of the image can be achieved via a dot product with a 2 × 2 matrix
  - S = [[horizontal_factor, 0], [0, vertical_factor]]



- Linear transform:
  - A dot product with an arbitrary matrix implements a linear transform.
- Scaling and rotation, listed previously, are by definition linear transforms

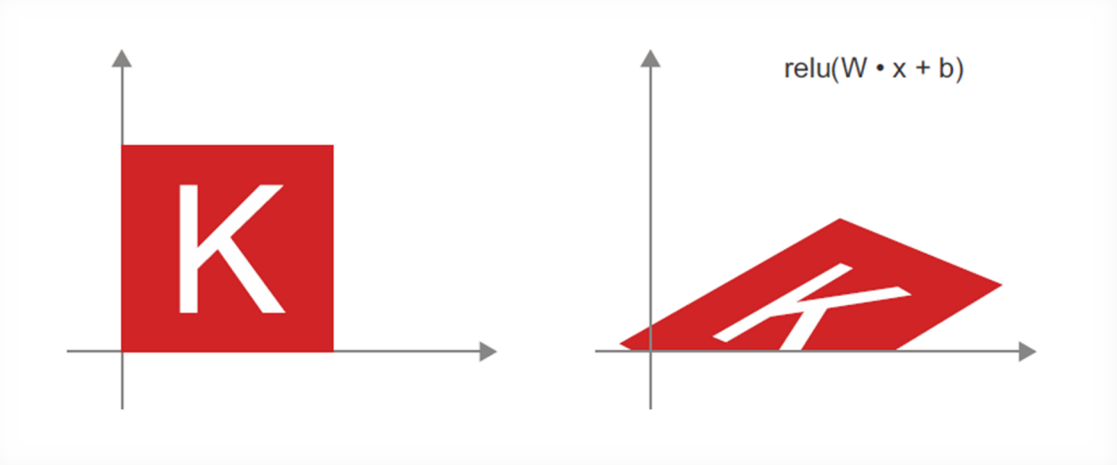
Geometric Interpretation of Tensor Operations

- Affine transform:
  - An affine transform is the combination of a linear transform (achieved via a dot product with some matrix) and a translation (achieved via a vector addition)
  - y = W • x + b computation implemented by the Dense layer
  - A Dense layer without an activation function is an affine layer



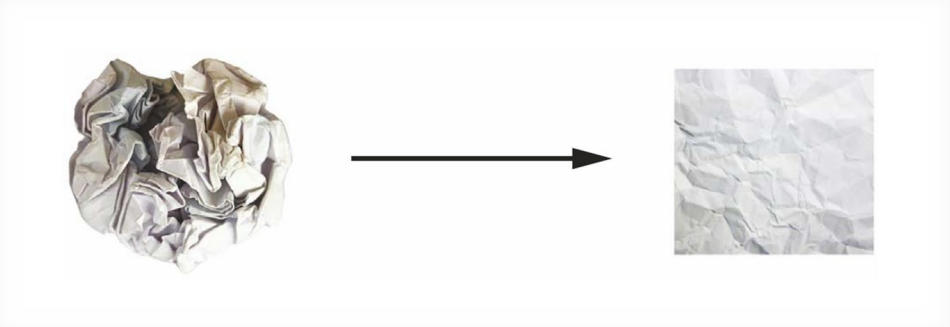
Geometric Interpretation of Tensor Operations

- Dense layer with relu activation:
  - An important observation about affine transforms is that if you apply many of them repeatedly, you still end up with an affine transform
  - affine2(affine1(x)) = W2 • (W1 • x + b1) + b2 = (W2 • W1) • x + (W2 • b1 + b2)
  - An affine transform where the linear part is the matrix W2 • W1 and the translation part is the vector W2 • b1 + b2
  - As a consequence, a multilayer neural network made entirely of Dense layers without activations would be equivalent to a single Dense layer

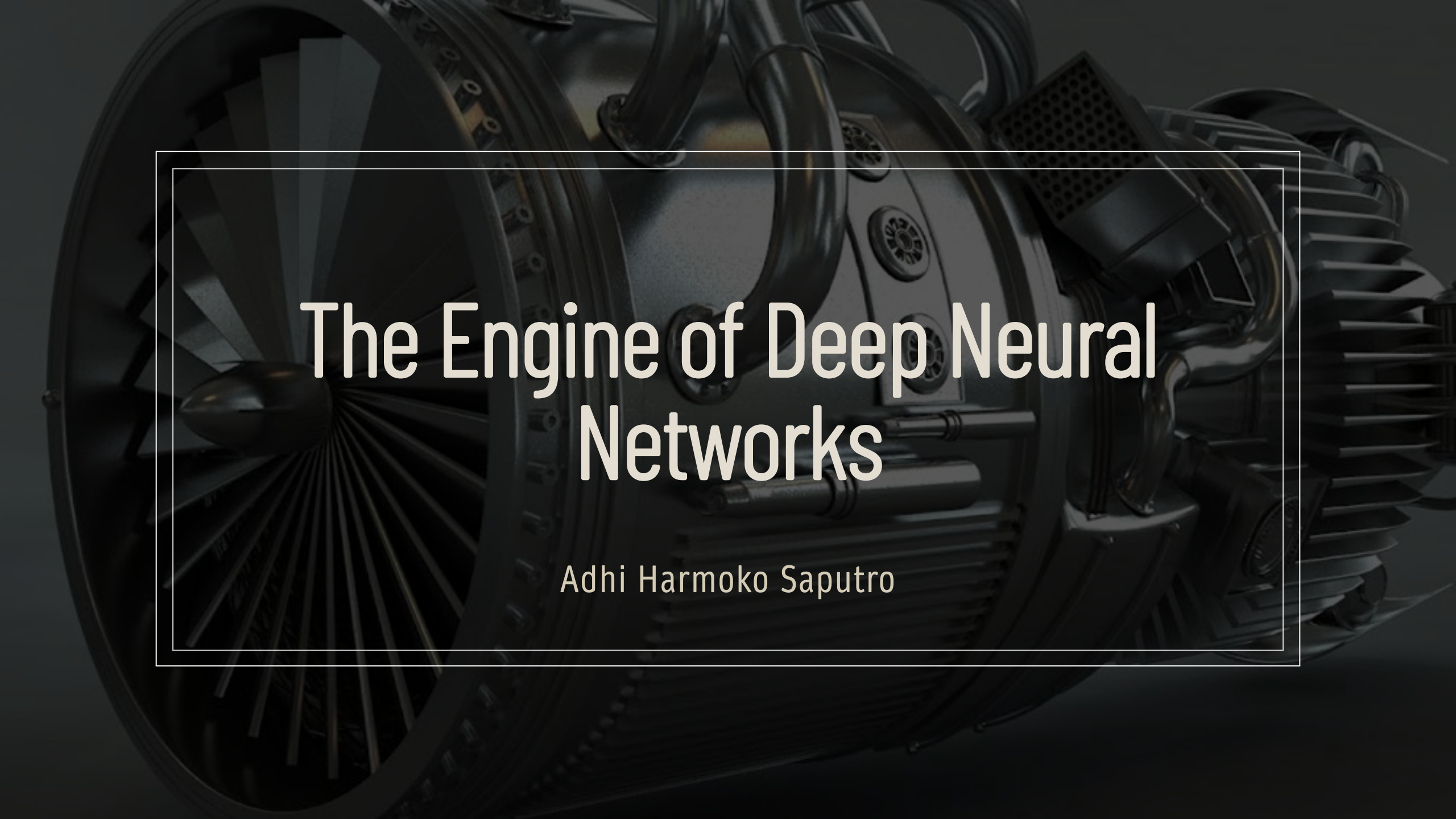


A Geometric Interpretation of Deep Learning

- Neural networks consist entirely of chains of tensor operations
  - The tensor operations are just simple geometric transformations of the input data
  - A neural network is a very complex geometric transformation in a high-dimensional space, implemented via a series of simple steps
- Uncrumpling a complicated manifold of data
  - A neural network: figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again



  - Deep learning: implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time



The Engine of Deep Neural Networks

Adhi Harmoko Saputro

Neural Network Training

- Each neural layer from the model example transforms its input data

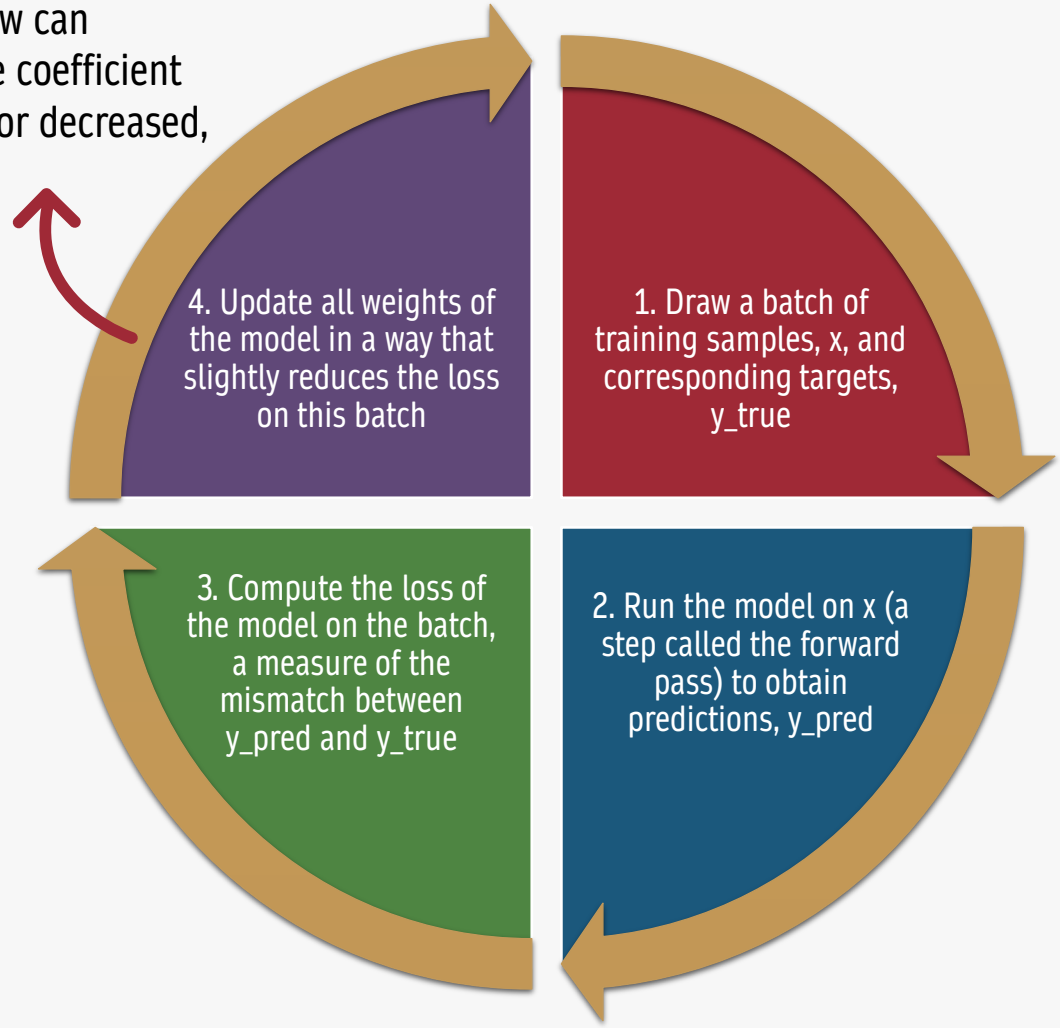
$$\texttt{output} = \texttt{relu}(\texttt{dot}(\texttt{input}, \texttt{W}) + \texttt{b})$$

  - W and b are tensors that are attributes of the layer (the kernel and bias attributes)
  - The weights or trainable parameters of the layer
  - Initially, these weight matrices are filled with small random values (random initialization)
  - Neural Network training: gradual adjustment based on a feedback signal of the information by the model from exposure to training data

Training Loop

Repeat following steps in a loop, until the loss seems sufficiently low

The difficult part: How can compute whether the coefficient should be increased or decreased, and by how much?



Deep Learning Optimizer

- Optimizers are algorithms or methods used to change the attributes of your neural network such as weights, bias and learning rate in order to reduce the losses

- Optimization algorithms or strategies are responsible for reducing the losses and to provide the most accurate results possible

Various Optimizers

- Gradient Descent
- Stochastic Gradient Descent
- Stochastic Gradient descent with momentum
- Mini-Batch Gradient Descent
- Adagrad
- RMSProp
- AdaDelta
- ADAM

Gradient Descent

- Gradient descent is the optimization technique that powers modern neural networks
  - All of the functions used in the models (such as dot or +) transform their input in a smooth and continuous way

- Example:

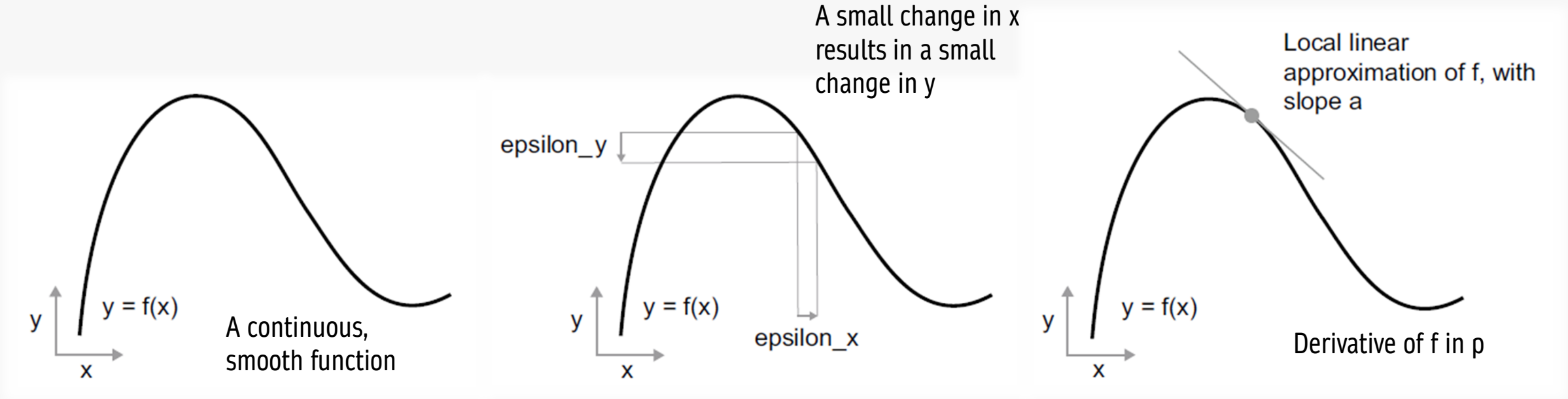
$$z = x + y$$

  - A small change in y only results in a small change in z
  - If the direction of the change in y was known, can infer the direction of the change in z

- Solved using mathematical operator called the gradient
  - Describe how the loss varies as you move the model's coefficients in different directions
  - A small change in the model's coefficients results in a small, predictable change in the loss value
  - Use it to move the coefficients (all at once in a single update, rather than one at a time) in a direction that decreases the loss

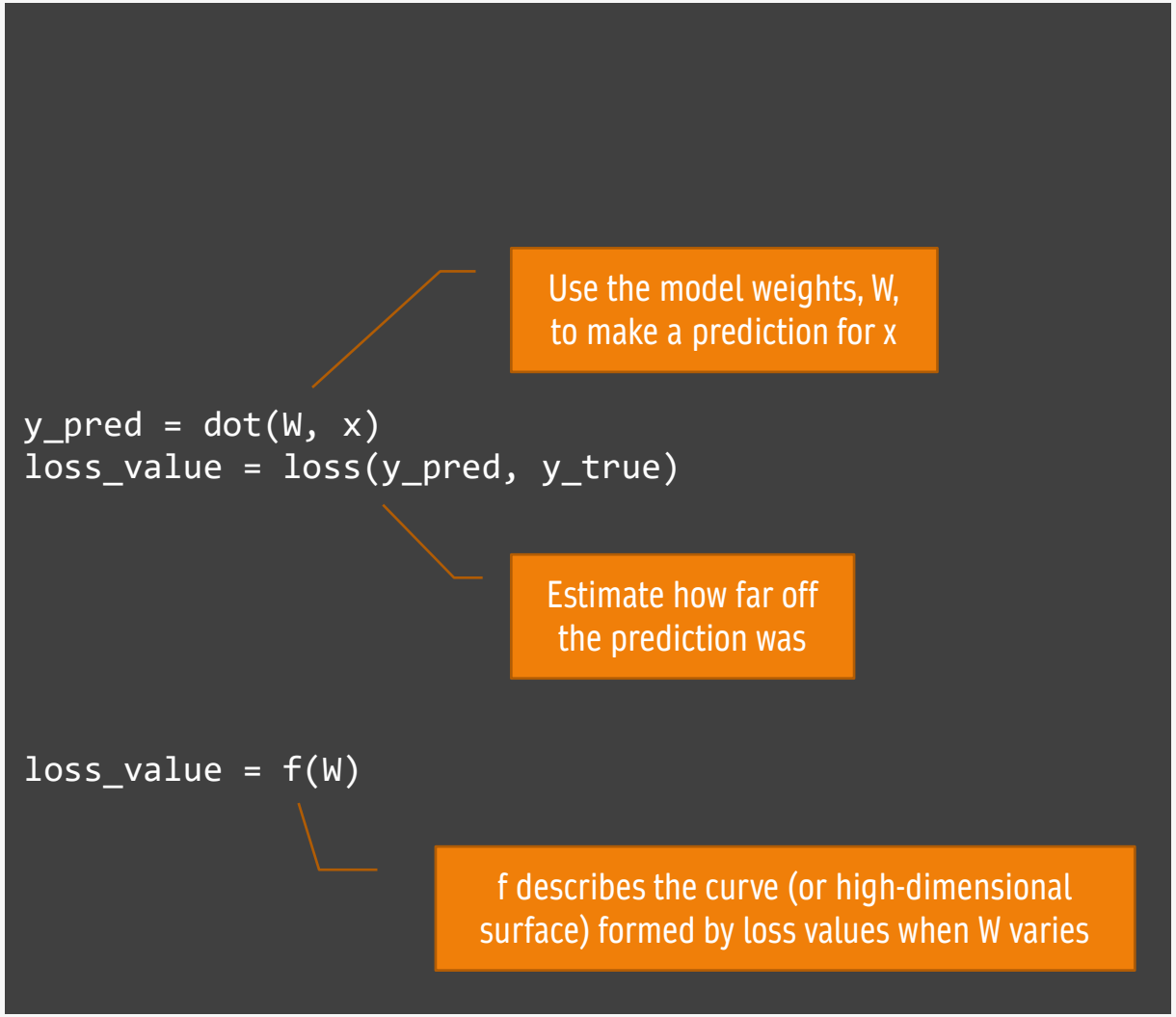
Derivative of a Tensor Operation: The Gradient

- A gradient:
  - the derivative of a tensor operation (or tensor function)
  - The generalization of the concept of derivatives to functions that take tensors as inputs
  - Represents the curvature of the multidimensional surface described by the function
  - Characterizes how the output of the function varies when its input parameters vary



The Gradient Example

- Consider
  - An input vector, x (a sample in a dataset)
  - A matrix, W (the weights of a model)
  - A target, y_true (what the model should learn to associate to x)
  - A loss function, loss (meant to measure the gap between the model's current predictions and y_true)
- Use W to compute a target candidate y_pred, and then compute the loss
  - How to use gradients to figure out how to update W so as to make loss_value smaller?
- Interprete as a function mapping values of W (the model's weights) to loss values:



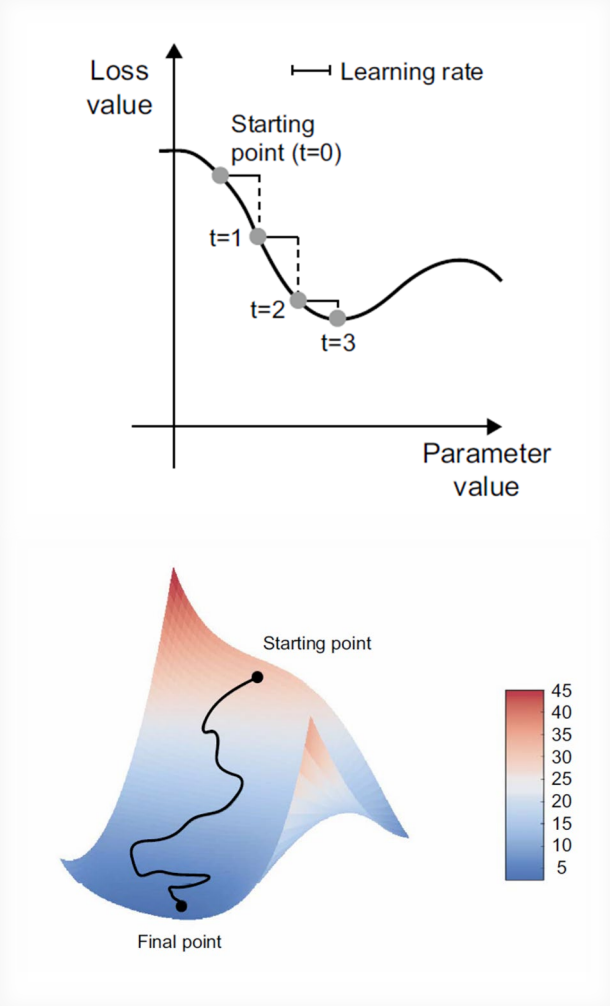
The Gradient Example

- If the current value of W is W0,
  - The derivative of f at the point W0 is a tensor grad(loss_value, W0), with the same shape as W
  - Each coefficient grad(loss_value, W0)[i, j] indicates the direction and magnitude of the change in loss_value you observe when modifying W0[i, j]
- The tensor grad(loss_value, W0) is the gradient of the function f(W) = loss_value in W0
  - Called as gradient of loss_value with respect to W around W0
  - The tensor describing the direction of steepest ascent of loss_value = f(W) around W0
- Reduce loss_value = f(W) by moving W in the opposite direction from the gradient
  - The direction of steepest ascent of f, which intuitively should put you lower on the curve
  - The scaling factor step is needed because grad(loss_value, W0) only approximates the curvature when close to W0

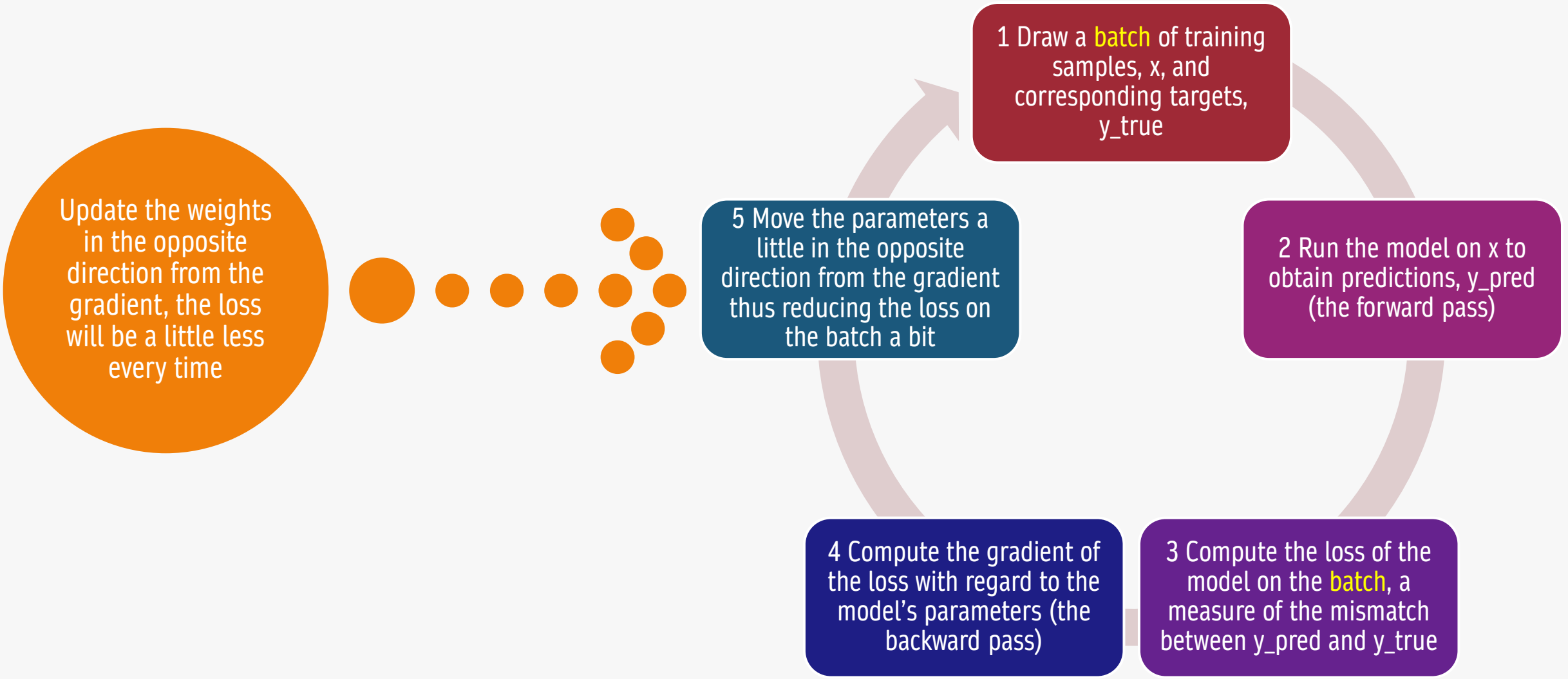
Stochastic Gradient Descent

- Stochastic gradient descent: finding analytically the combination of weight values that yields the smallest possible loss function
  - The term stochastic refers to the fact that each batch of data is drawn at random (stochastic is a scientific synonym of random)

- Intuitively it's important to pick a reasonable value for the learning_rate factor
  - If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum
  - If too large, your updates may end up taking you to completely random locations on the curve

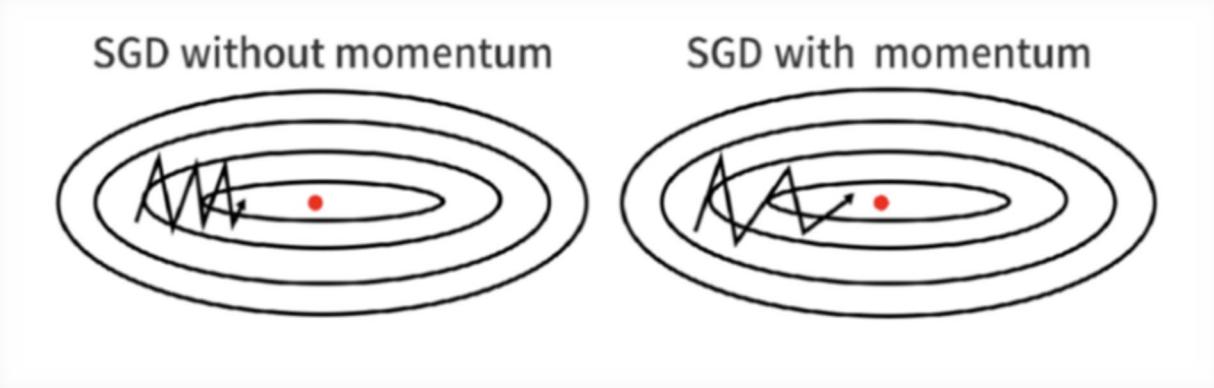


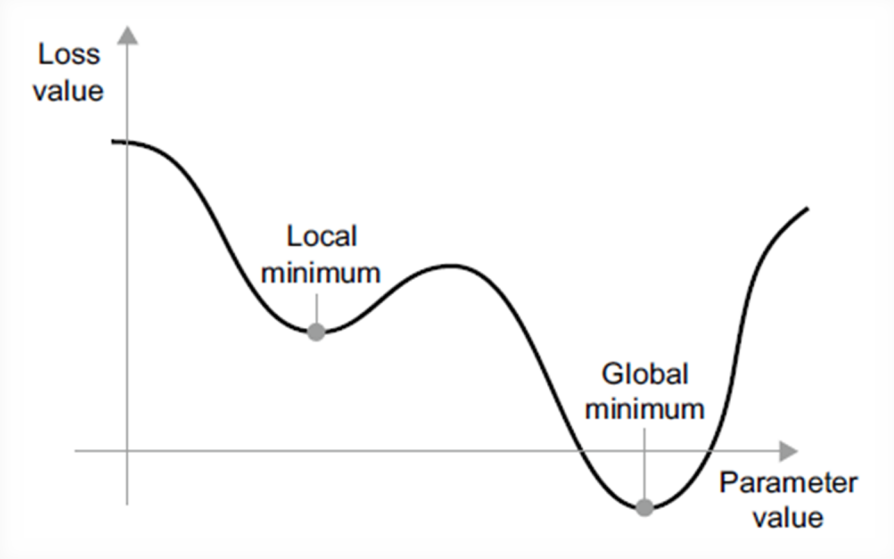
Mini-batch SGD



Stochastic Gradient Descent with Momentum

- SGD with Momentum is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent
  - Momentum simulates the inertia of an object when it is moving
  - The direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction
  - In this way, can increase the stability to a certain extent, so that can learn faster, and also have the ability to get rid of local optimization





AdaGrad (Adaptive Gradient Descent)

- An extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) seen over the course of the search

- The intuition behind AdaGrad:
  - Use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations

RMS Prop (Root Mean Square)

- RMS-Prop is a special version of AdaGrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients
- RMS-Prop basically combines momentum with AdaGrad

- Advantages of RMSProp
  - It is a very robust optimizer which has pseudo-curvature information
  - It can deal with stochastic objectives very nicely, making it applicable to mini batch learning
  - It converges faster than momentum
- Disadvantages of RMSProp
  - Learning rate is still manual, because the suggested value is not always appropriate for every task

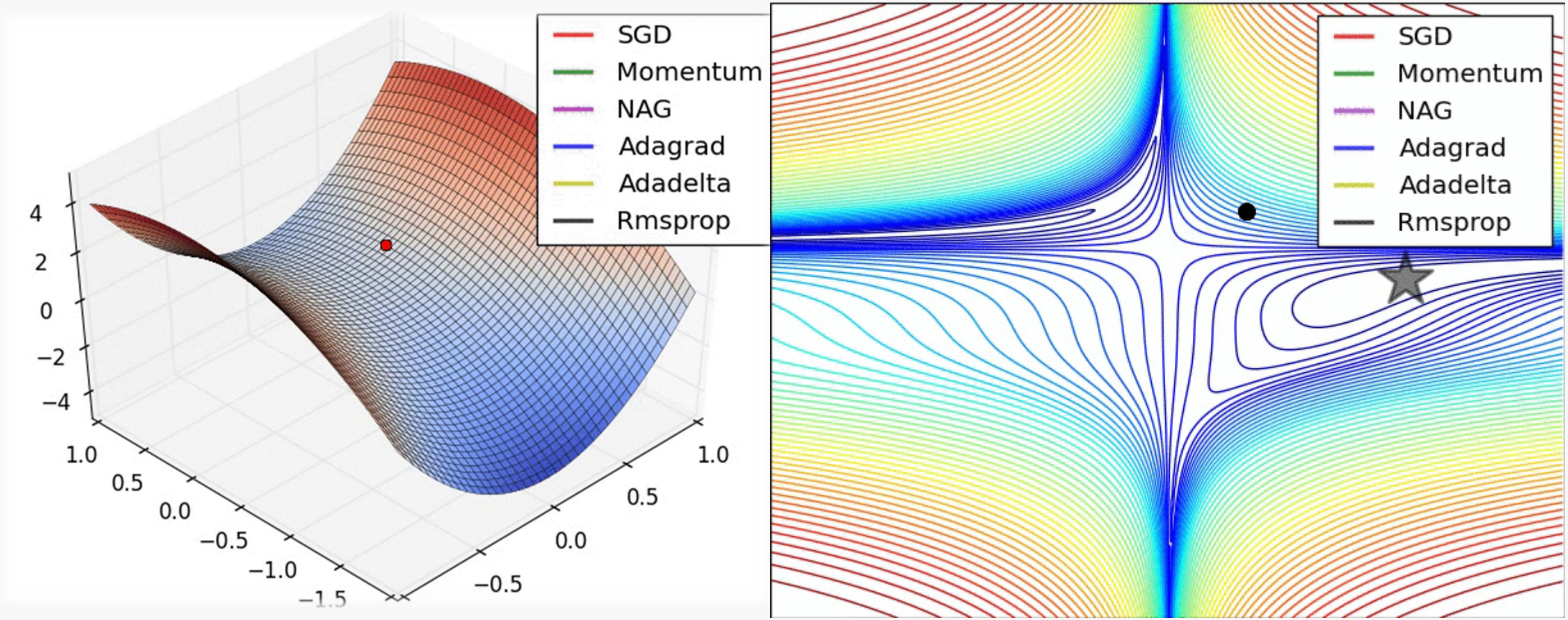
AdaDelta

- Adadelta is an extension of AdaGrad
  - Tries to reduce AdaGrad's aggressive, monotonically reducing the learning rate and remove decaying learning rate problem
  - Do not need to set the default learning rate as take the ratio of the running average of the previous time steps to the current gradient

ADAM (Adaptive Moment Estimation)

- This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training
  - Adam optimizer updates the learning rate for each network weight individually
  - Inherit the features of both AdaGrad and RMSprop algorithms, also extensions of the stochastic gradient descent algorithms
  - Uses the second moment of the gradients
  - Mean the uncentred variance by the second moment of the gradients

Comparing the Optimizer



Source: https://awesomeopensource.com/project/Jaewan-Yun/optimizer-visualization

How to Choose Optimizers?

If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam

RMSprop, Adadelta, Adam have similar effects in many cases

Adam just added bias-correction and momentum on the basis of RMSprop

As the gradient becomes sparse, Adam will perform better than RMSprop

Backpropagation Algorithm

- Backpropagation Algorithm
  - To compute the gradient of complex expressions in practice
  - To get the gradient of the loss with regard to the weights
  - A way to use the derivatives of simple operations (such as addition, relu, or tensor product) to easily compute the gradient of arbitrarily complex combinations of these atomic operations

- The Chain Rule

```
grad(y,x) == grad(y,x1) * grad(x1,x)
```

  - Enables to compute the derivative of fg as long as you know the derivatives of f and g

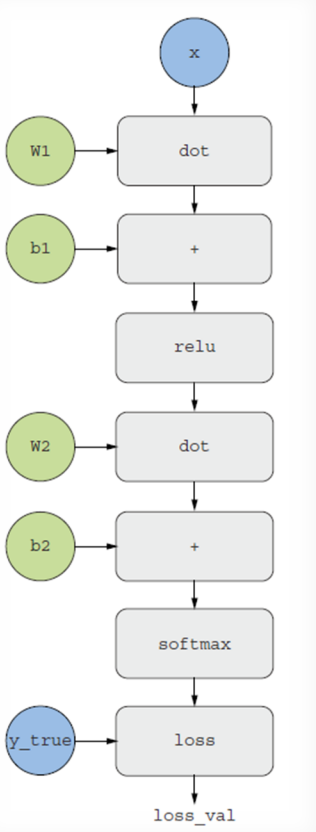
```
grad(y,x) == grad(y,x3)*grad(x3,x2)*grad(x2,x1)*grad(x1,x)
```

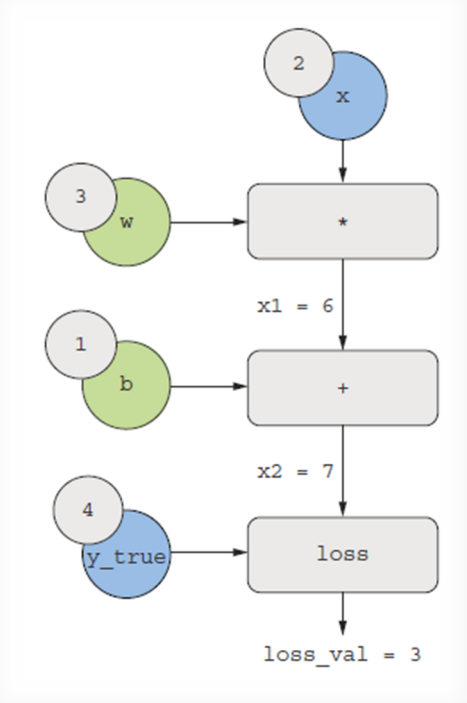
```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y  = f(x3)
    return y
```

Computation Graph Representation of Two-layer Model

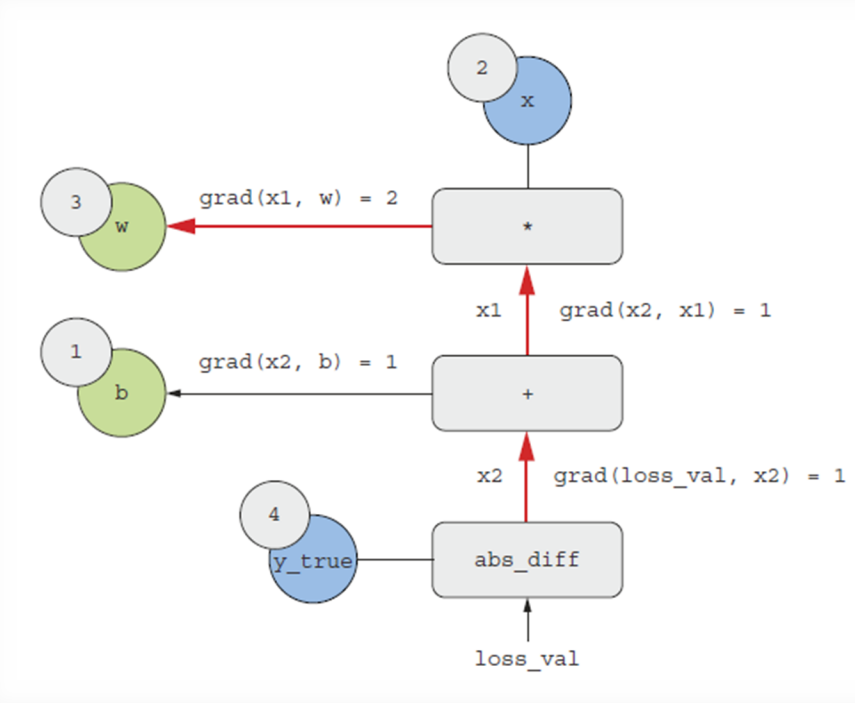
- A computation graph is the data structure at the heart of TensorFlow and the deep learning revolution in general
- Compute the gradient of arbitrarily complex combinations of these atomic operations

$$\texttt{loss\_value} = \texttt{loss}\left(\texttt{y\_true}, \texttt{softmax}\left(\texttt{dot}\left(\texttt{relu}\left(\texttt{dot}\left(\texttt{inputs}, \texttt{W1}\right) + \texttt{b1}\right), \texttt{W2}\right) + \texttt{b2}\right)\right)$$





Running a forward pass



Running a backward pass

Automatic Differentiation

- Backpropagation is simply the application of the chain rule to a computation graph
  - Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, computing the contribution that each parameter had in the loss value

- Automatic Differentiation
  - Use to implement neural networks in modern frameworks, such as TensorFlow
  - Makes it possible to retrieve the gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down the forward pass
  - Never have to implement backpropagation yourself

The Gradient Tape in Tensorflow

- The GradientTape: TensorFlow's powerful automatic differentiation capabilities
  - "Record" the tensor operations that run inside it, in the form of a computation graph (sometimes called a "tape")
  - Retrieve the gradient of any output with respect to any variable or set of variables (instances of the tf.Variable class)
  - A tf.Variable is a specific kind of tensor meant to hold mutable state

```
import tensorflow as tf

x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3

grad_of_y_wrt_x = tape.gradient(y, x)
```

Instantiate a scalar Variable with an initial value of 0

Open a GradientTape scope

Use the tape to retrieve the gradient of the output y with respect to our variable x

- The GradientTape works with tensor operations

Instantiate a Variable with shape (2, 2) and an initial value of all zeros

grad_of_y_wrt_x is a tensor of shape (2, 2) (like x) describing the curvature of y = 2 * a + 3 around x = [[0, 0], [0, 0]]

```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

The Gradient Tape in Tensorflow

- It also works with lists of variables

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))

with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b

grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

matmul is “dot product” in TensorFlow

grad_of_y_wrt_W_and_b is a list of two tensors with the same shapes as W and b

Understanding TensorFlow & Core Keras APIs

Badan Meteorologi, Klimatologi, dan Geofisika
22 – 31 Desember 2021

First Steps with TensorFlow

Low-level Tensor Manipulation

High-level Deep Learning Concepts

First Steps with TensorFlow

- Low-level Tensor Manipulation: The infrastructure that underlies all modern machine learning
  - Tensors, including special tensors that store the network's state (variables)
  - Tensor operations such as addition, relu, matmul
  - Backpropagation, a way to compute the gradient of mathematical expressions (handled in TensorFlow via the GradientTape object)
- High-level Deep Learning Concepts
  - Layers, which are combined into a model
  - A loss function, which defines the feedback signal used for learning
  - An optimizer, which determines how learning proceeds
  - Metrics to evaluate model performance, such as accuracy
  - A training loop that performs mini-batch stochastic gradient descent

Constant Tensors & Variables

- Tensors need to be created with some initial value
  - Create all-ones or all-zeros tensors

```
import tensorflow as tf
x = tf.ones(shape=(2, 1))
print(x)
tf.Tensor( [[1.] [1.]], shape=(2, 1), dtype=float32)
```

Equivalent to np.ones(shape=(2, 1))

```
x = tf.zeros(shape=(2, 1))
print(x)
tf.Tensor( [[0.] [0.]], shape=(2, 1), dtype=float32)
```

Equivalent to

Constant Tensors & Variables

- Tensors of values drawn from a random distribution

  - Tensor of random values drawn from a normal distribution with mean 0 and standard deviation 1
  - Equivalent to np.random.normal(size=(3, 1), loc=0., scale=1.)

```
x = tf.random.normal(shape=(3, 1), mean=0., stddev=1.)
print(x)
tf.Tensor( [[-0.920044 ] [ 2.1811383] [-1.5889181]],
shape=(3, 1), dtype=float32)
```

  - Tensor of random values drawn from a uniform distribution between 0 and 1
  - Equivalent to np.random.uniform(size=(3, 1), low=0., high=1.)

```
x = tf.random.uniform(shape=(3, 1), minval=0., maxval=1.)
print(x)
tf.Tensor( [[0.93827045] [0.30227625] [0.60100555]],
shape=(3, 1), dtype=float32)
```

Constant Tensors & Variables: Assignment

- A significant difference between NumPy arrays and TensorFlow tensors is
  - NumPy arrays are assignable

```
import numpy as np
x = np.ones(shape=(2, 2))
x[0, 0] = 0.
```

  - Tensor-Flow tensors aren't assignable

```
import tensorflow as tf
x = tf.ones(shape=(2, 2))
x[0, 0] = 0.
---------------------------------------------------------------
------------------ TypeError Traceback (most recent
call last)
~\AppData\Local\Temp/ipykernel_21340/1748468457.py in
<module> 1 import tensorflow as tf 2 x =
tf.ones(shape=(2, 2)) ----> 3 x[0, 0] = 0. TypeError:
'tensorflow.python.framework.ops.EagerTensor' object
does not support item assignment
```

- To train a model, need to update its state, which is a set of tensors
  - Due to tensors aren't assignable use tf.Variable that is the class meant to manage modifiable state in TensorFlow

Constant Tensors & Variables: Assignment

- To create a variable, need to provide some initial value, such as a random tensor

```
v = tf.Variable(initial_value=tf.random.normal(shape=(3, 1)))
print(v)
```

- The state of a variable can be modified via its assign method
  - It also works for a subset of the coefficients

```
v.assign(tf.ones((3, 1)))
print(v)
```

- Assigning a value to a subset of a TensorFlow variable
  - assign_add() and assign_sub() are efficient equivalents of += and -=

```
v[0, 0].assign(3.)
print(v)

v.assign_add(tf.ones((3, 1)))
print(v)
```

Doing Math in TensorFlow

- TensorFlow offers a large collection of tensor operations to express mathematical formulas
  - Each of the preceding operations gets executed on the fly
  - At any point, can print what the current result is, just like in NumPy
  - Call it as eager execution

- Eager Execution vs. Graph Execution in TensorFlow: https://towardsdatascience.com/eager-execution-vs-graph-execution-which-is-better-38162ea4dbf6

```
a = tf.ones((2, 2))
b = tf.square(a)          # Take the square
c = tf.sqrt(a)            # Take the square root
d = b + c                 # Add two tensors (element-wise)
e = tf.matmul(a, b)       # Take the product of two tensors
e *= d                    # Multiply two tensors (element-wise)

print(a)
print(b)
print(c)
print(d)
print(e)


tf.Tensor( [[1. 1.] [1. 1.]], shape=(2, 2), dtype=float32)
tf.Tensor( [[1. 1.] [1. 1.]], shape=(2, 2), dtype=float32)
tf.Tensor( [[1. 1.] [1. 1.]], shape=(2, 2), dtype=float32)
tf.Tensor( [[2. 2.] [2. 2.]], shape=(2, 2), dtype=float32)
tf.Tensor( [[4. 4.] [4. 4.]], shape=(2, 2), dtype=float32)
```

GradientTape API

- Retrieve the gradient of any differentiable expression with respect to any of its inputs
  - Apply some computation to one or several input tensors
  - Retrieve the gradient of the result with respect to the inputs

- Used to retrieve the gradients of the loss of a model with respect to its weights

```
gradients = tape.gradient(loss, weights)
```

- It's necessary
  - It would be too expensive to preemptively store the information required to compute the gradient of anything with respect to anything
  - To avoid wasting resources, the tape needs to know what to watch
  - Trainable variables are watched by default because computing the gradient of a loss with regard to a list of trainable variables is the most common use of the gradient tape

```
input_var = tf.Variable(initial_value=3.)
with tf.GradientTape() as tape:
   result = tf.square(input_var)
gradient = tape.gradient(result, input_var)



input_const = tf.constant(3.)
with tf.GradientTape() as tape:
   tape.watch(input_const)
   result = tf.square(input_const)
gradient = tape.gradient(result, input_const)
```

GradientTape API

- The gradient tape is a powerful utility, even capable of computing second-order gradients
  - The gradient of a gradient

- Using nested gradient tapes to compute second-order gradients
  - Measure the position of a falling apple along a vertical axis over time and find that it verifies position(time) = 4.9 * time ** 2
    - What is its acceleration?

```
time = tf.Variable(0.)

with tf.GradientTape() as outer_tape:
    with tf.GradientTape() as inner_tape:
        position =  4.9 * time ** 2
    speed = inner_tape.gradient(position, time)
acceleration = outer_tape.gradient(speed, time)
```

- Use the outer tape to compute the gradient of the gradient from the inner tape
- Naturally, the answer is 4.9 * 2 = 9.8

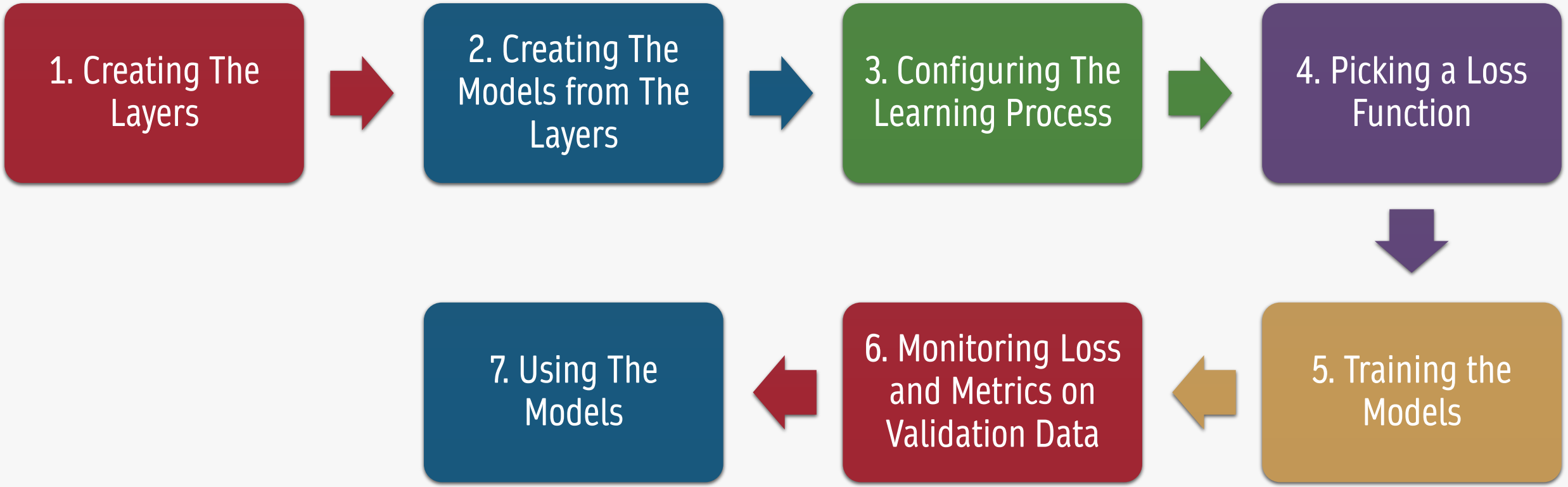
Understanding Core Keras APIs

Know the basics of TensorFlow

Can use it to implement a toy model from scratch

More productive, more robust path to deep learning: the Keras API

Understanding Core Keras APIs



1. Creating The Layers

- The fundamental data structure in neural networks is the layer
  - A layer is a data processing module that takes as input one or more tensors and that outputs one or more tensors
  - Some layers are stateless, but more frequently layers have a state: the layer's weights
    - One or several tensors learned with stochastic gradient descent, which together contain the network's knowledge
- Different types of layers are appropriate for different tensor formats and different types of data processing
  - Simple vector data, stored in rank-2 tensors of shape (samples, features), is often processed by densely connected layers, also called fully connected or dense layers (the Dense class in Keras)
  - Sequence data, stored in rank-3 tensors of shape (samples, timesteps, features), is typically processed by recurrent layers, such as an LSTM layer, or 1D convolution layers (Conv1D)
  - Image data, stored in rank-4 tensors, is usually processed by 2D convolution layers (Conv2D)
- Think of layers as the LEGO bricks of deep learning, a metaphor
  - Made explicit by Keras
  - Building deep learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

The Base Layer Class in Keras

- Everything in Keras is either a Layer or something that closely interacts with a Layer
- A Layer is an object that encapsulates some state (weights) and some computation (a forward pass)
  - The weights are typically defined in a build()
  - Could also be created in the constructor, __init__()
  - The computation is defined in the call() method
- Example: A Dense layer implemented as a Layer subclass
- Don't worry if you don't understand the purpose of these build() and call()

```python
from tensorflow import keras

class SimpleDense(keras.layers.Layer):

    def __init__(self, units, activation=None):
        super().__init__()
        self.units = units
        self.activation = activation

    def build(self, input_shape):
        input_dim = input_shape[-1]
        self.W = self.add_weight(shape=(input_dim, self.units),
                                 initializer="random_normal")
        self.b = self.add_weight(shape=(self.units,),
                                 initializer="zeros")

    def call(self, inputs):
        y = tf.matmul(inputs, self.W) + self.b
        if self.activation is not None:
            y = self.activation(y)
        return y
```

Weight creation takes place in the build() method

add_weight() is a shortcut method for creating weights

define the forward pass computation in the call() method

The Base Layer Class in Keras

- Once instantiated, a layer like this can be used just like a function
  - Taking as input a TensorFlow tensor

Instantiate our layer, defined previously

Create some test inputs

Call the layer on the inputs, just like a function

```
my_dense = SimpleDense(units=32, activation=tf.nn.relu)
input_tensor = tf.ones(shape=(2, 784))
output_tensor = my_dense(input_tensor)
print(output_tensor.shape)
(2, 32)
```

Building Layers on The Fly

- Just like with LEGO bricks
  - Can only “clip” together layers that are compatible
  - The notion of layer compatibility here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape

- Example: A dense layer with 32 output units
  - The layer will return a tensor where the first dimension has been transformed to be 32
  - It can only be connected to a downstream layer that expects 32-dimensional vectors as its input

```
from tensorflow.keras import layers

layer = layers.Dense(32, activation="relu")
```

Building Layers on The Fly

- In Keras, don't worry about size compatibility most of the time
  - The layers that add to the models are dynamically built to match the shape of the incoming layer
  - The layers didn't receive any information about the shape of their inputs, automatically inferred their input shape as being the shape of the first inputs they see

```
from tensorflow.keras import models
from tensorflow.keras import layers

model = models.Sequential([
    layers.Dense(
        32,
        activation="relu"),
    layers.Dense(32)
])
```

Building Layers on The Fly

- If have the Dense layer named NaiveDense
  - That's not ideal, because it would lead to models that each new layer needs to be made aware of the shape of the layer before it

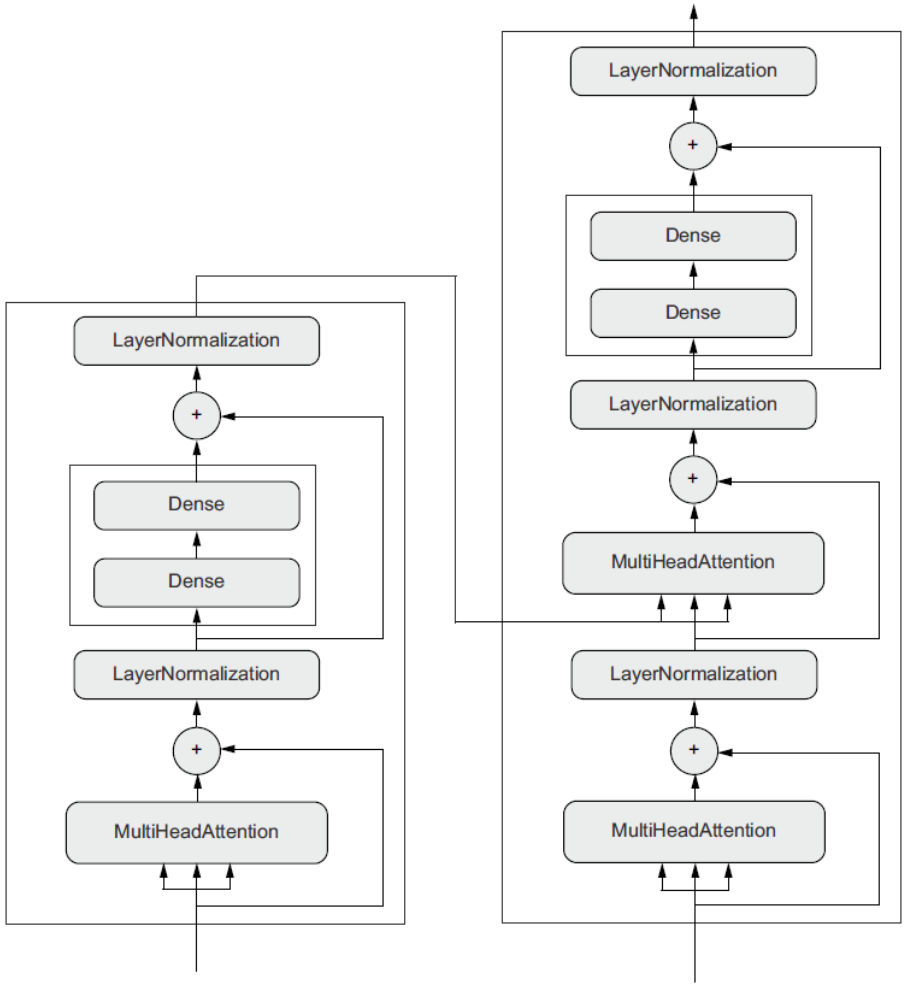
```
model = NaiveSequential([
    NaiveDense(input_size=784, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=64, activation="relu"),
    NaiveDense(input_size=64, output_size=32, activation="relu"),
    NaiveDense(input_size=32, output_size=10, activation="softmax")
    ])
```

- If reimplement the NaiveDense layer as a Keras layer capable of automatic shape inference
  - it would look like the previous SimpleDense layer
  - With automatic shape inference, our previous example becomes simple and neat

```
model = keras.Sequential([
    SimpleDense(32, activation="relu"),
    SimpleDense(64, activation="relu"),
    SimpleDense(32, activation="relu"),
    SimpleDense(10, activation="softmax")
    ])
```

2. Creating The Models from The Layers

- A deep learning model is a graph of layers (the Model class in Keras)
- Much broader variety of network topologies
  - Two-branch networks
  - Multihead networks
  - Residual connections
- Two ways of building models in Keras
  - Directly subclass the Model class
  - Use the Functional API (less code)



The Transformer Architecture

A Network Topology

- The topology of a model defines a hypothesis space
  - Constrain the space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data
  - Searching for is a good set of values for the weight tensors involved in these tensor operations
  - Encodes the assumptions about the problem, the prior knowledge that the model starts with

- Picking the right network architecture is more an art than a science
  - Although there are some best practices and principles that can rely on, only practice can help to become a proper neural-network architect

3. Configuring The Learning Process

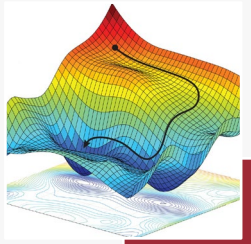
- Loss function (objective function)
  - The quantity that will be minimized during training
  - Represents a measure of success for the task at hand
- Optimizer
  - Determines how the network will be updated based on the loss function
  - Implements a specific variant of stochastic gradient descent (SGD)
- Metrics
  - The measures of success you want to monitor during training and validation, such as classification accuracy
  - Training will not optimize directly for these metrics
  - Metrics don’t need to be differentiable

Configuring The Learning Process

- The compile() method configures the training process
  - picked the loss, optimizer, and metrics
- Passed the optimizer, loss, and metrics as strings
  - "rmsprop“ : keras.optimizers.RMSprop()
- Passing a learning_rate argument to the optimizer with custom losses or metrics

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(
    optimizer="rmsprop",
    loss="mean_squared_error",
    metrics=["accuracy"])


model.compile(
    optimizer=keras.optimizers.RMSprop(),
    loss=keras.losses.MeanSquaredError(),
    metrics=[keras.metrics.BinaryAccuracy()])



model.compile(
    optimizer=keras.optimizers.RMSprop(learning_rate=1e-4),
    loss=my_custom_loss,
    metrics=[my_custom_metric_1, my_custom_metric_2])
```

Keras Optimizers, Losses & Metrics



Optimizers:

- SGD (with or without momentum)
- RMSprop
- Adam
- Adagrad

Losses:

- CategoricalCrossentropy
- SparseCategoricalCrossentropy
- BinaryCrossentropy
- MeanSquaredError
- KLDivergence
- CosineSimilarity

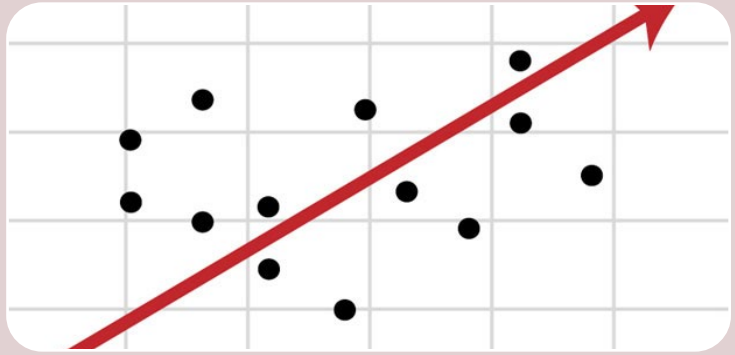
Metrics:

- CategoricalAccuracy
- SparseCategoricalAccuracy
- BinaryAccuracy
- AUC
- Precision
- Recall

4. Picking a Loss Function

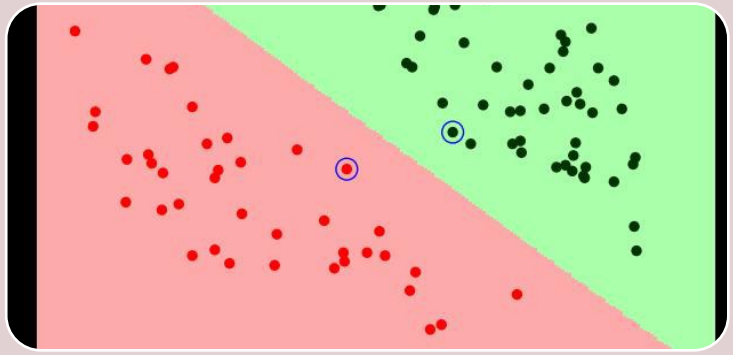
- Choosing the right loss function for the right problem is extremely important
- There are simple guidelines you can follow to choose the correct loss for common problems such as classification, regression, and sequence prediction
  - Use binary crossentropy for a two-class classification problem
  - Use categorical crossentropy for a many-class classification problem

Keras Loss Function



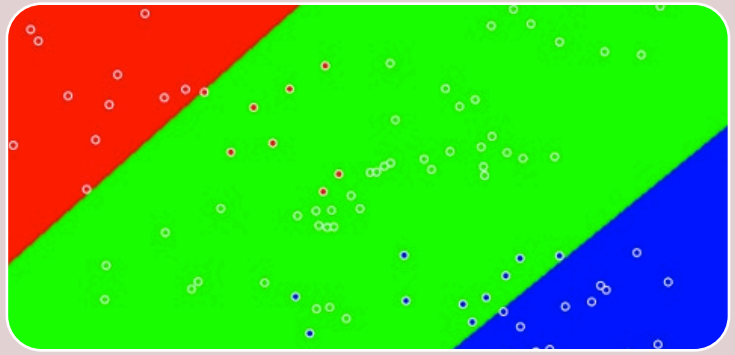
Regression Loss Functions

- Mean Squared Error Loss
- Mean Squared Logarithmic Error Loss
- Mean Absolute Error Loss



Binary Classification Loss Functions

- Binary Cross-Entropy
- Hinge Loss
- Squared Hinge Loss



Multi-Class Classification Loss Functions

- Multi-Class Cross-Entropy Loss
- Sparse Multiclass Cross-Entropy Loss
- Kullback Leibler Divergence Loss

5. Training the Models

- The data (inputs and targets) to train on
  - Passed either in the form of NumPy arrays or a TensorFlow Dataset object
- The number of epochs to train for
  - How many times the training loop should iterate over the data passed
- The batch size to use within each epoch of mini-batch gradient descent
  - The number of training examples considered to compute the gradients for one weight update step

5. Training the Models

- Calling fit() with NumPy data

- The history object contains a history field
  - A dict mapping keys such as "loss" or specific metric names to the list of their perepoch values

```
history = model.fit(
    inputs,
    targets,
    epochs=5,
    batch_size=128
)

history.history
```

The input examples, as a NumPy array

The corresponding training targets, as a NumPy array

The training loop will iterate over the data 5 times

The training loop will iterate over the data in batches of 128 examples

6. Monitoring Loss and Metrics on Validation Data

- The goal of machine learning is not to obtain models that perform well on the training data
  - The goal is to obtain models that perform well in general, and particularly on data points that the model has never encountered before

- Standard practice to reserve a subset of the training data as validation data
  - Won't be training the model on this data
  - Use it to compute a loss value and metrics value
  - Do this by using the validation_data argument in fit()

Using The validation_data Argument

- The validation data could be passed as NumPy arrays or as a TensorFlow Dataset object
- The value of the loss on the validation data is called the “validation loss,” to distinguish it from the “training loss.”

```
model = keras.Sequential([keras.layers.Dense(1)])
model.compile(optimizer=keras.optimizers.RMSprop(learning_rate=0.1),
              loss=keras.losses.MeanSquaredError(),
              metrics=[keras.metrics.BinaryAccuracy()])

indices_permutation = np.random.permutation(len(inputs))
shuffled_inputs = inputs[indices_permutation]
shuffled_targets = targets[indices_permutation]

num_validation_samples = int(0.3 * len(inputs))
val_inputs = shuffled_inputs[:num_validation_samples]
val_targets = shuffled_targets[:num_validation_samples]
training_inputs = shuffled_inputs[num_validation_samples:]
training_targets = shuffled_targets[num_validation_samples:]
model.fit(
    training_inputs,
    training_targets,
    epochs=5,
    batch_size=16,
    validation_data=(val_inputs, val_targets)
)
```

To avoid having samples from only one class in the validation data, shuffle the inputs and targets using a random indices permutation

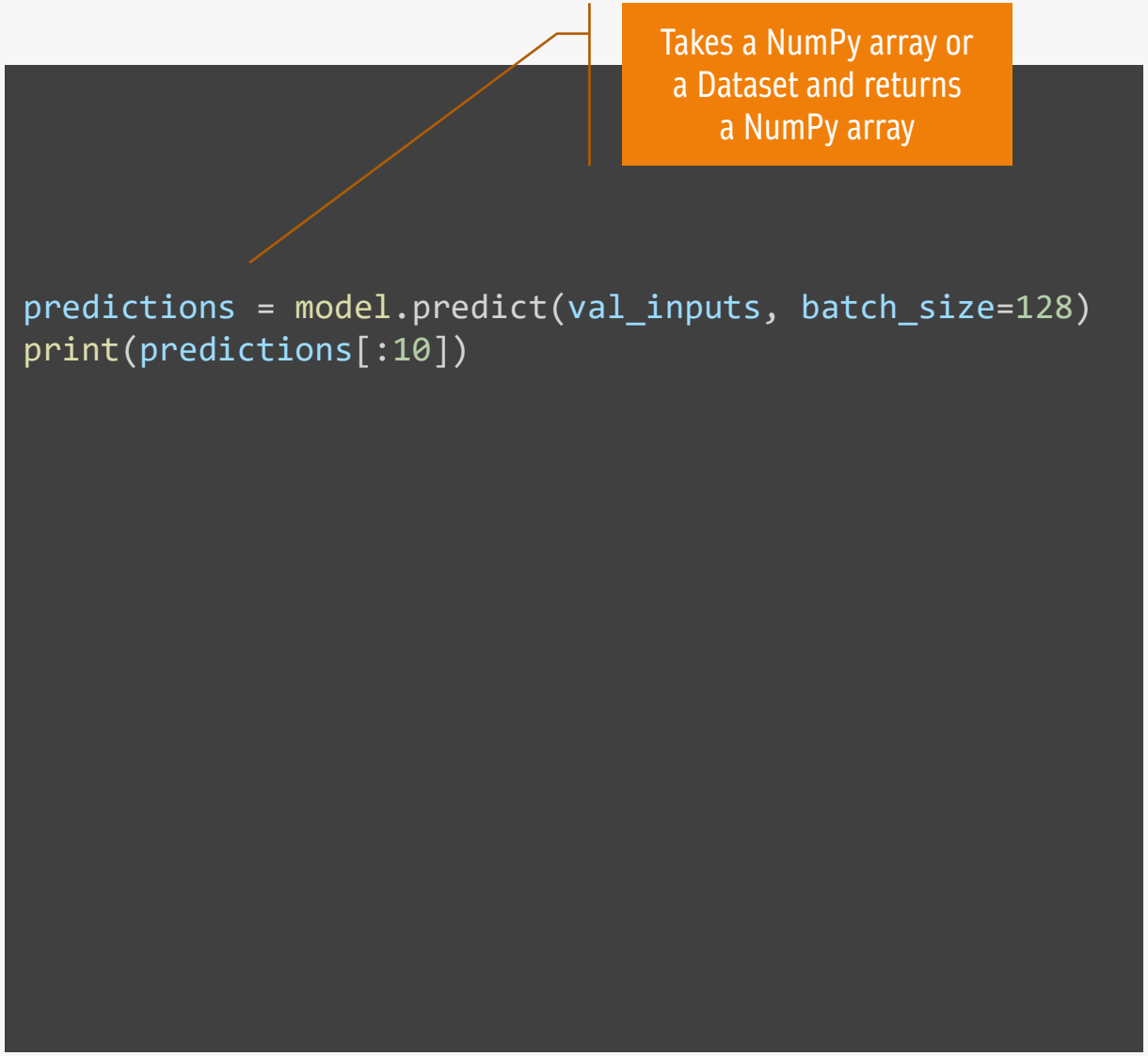
Reserve 30% of the training inputs and targets for validation (we’ll exclude these samples from training and reserve them to compute the validation loss and metrics)

Validation data, used only to monitor the validation loss and metrics

7. Using The Models

- Use to make predictions on new data (inference)
  - A naive approach would simply be to __call__() the model

- A better way to do inference is to use the predict() method
  - Iterate over the data in small batches and return a NumPy array of predictions



THANK YOU!

Adhi Harmoko Saputro