1. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?

No, it is not recommended to initialize all the weights in a neural network to the same value, even if the value is selected randomly using He initialization. While random initialization is important to break symmetry and avoid convergence issues, initializing all the weights to the same value can still lead to problems.

When all the weights are set to the same value, it essentially means that all the neurons in a given layer are receiving the same input. As a result, during the forward and backward propagation steps of training, the gradients will be the same for all the weights in that layer. This symmetry in weight updates can cause a lack of diversity in the learning process, limiting the network's capacity to learn and represent complex relationships in the data.

1. Is it OK to initialize the bias terms to 0?

Yes, it is generally acceptable to initialize the bias terms to 0 in neural networks. Initializing the biases to 0 is a common practice in many cases.

The bias term is an additional parameter in each neuron of a neural network that allows for the independent adjustment of the neuron's output. It provides flexibility in shifting the activation function, enabling the network to learn better representations and improve performance.

1. Name three advantages of the SELU activation function over ReLU.

Self-Normalizing Property

Smooth Activation

Improved Learning in Deep Networks

1. In which cases would you want to use each of the following activation functions: SELU, leaky ReLU (and its variants), ReLU, tanh, logistic, and softmax?

Different activation functions serve different purposes based on the characteristics of the problem and the network architecture. Here's a summary of the common use cases for each of the activation functions you mentioned:

SELU (Scaled Exponential Linear Unit):

Use SELU when building deep neural networks that require stable training and maintain signal magnitudes. SELU helps prevent vanishing or exploding gradients and supports self-normalization in deep networks. It is particularly useful in architectures with many layers.

Use Leaky ReLU and its variants when dealing with the "dying ReLU" problem, where some neurons in ReLU-based networks become inactive and do not contribute to learning. These variants introduce a small negative slope for negative inputs, which allows gradients to flow during backpropagation and can prevent dead neurons.

Leaky ReLU and its variants (e.g., Parametric ReLU, Randomized ReLU):

ReLU is widely used as a default choice due to its simplicity and computational efficiency. Use ReLU in most cases, especially in shallow or deep architectures, for tasks such as image classification or object detection. However, be aware of the potential issue of dead neurons in certain scenarios, which can be addressed with leaky ReLU or its variants.

tanh (Hyperbolic Tangent):

Use tanh when you need an activation function that produces outputs between -1 and 1. It is often used in recurrent neural networks (RNNs) and certain types of autoencoders. tanh can model both positive and negative relationships and is symmetric around 0, making it useful for tasks that require capturing bipolar patterns. Logistic (Sigmoid):

Use logistic (sigmoid) when you need a smooth activation function that produces outputs between 0 and 1. It is commonly used in binary classification tasks where the goal is to estimate probabilities. However, note that the logistic function is rarely used in hidden layers of deep neural networks due to the vanishing gradient problem.

Softmax:

Use softmax in the output layer of a neural network when dealing with multi-class classification problems. Softmax outputs a probability distribution over multiple classes, ensuring that the predicted probabilities sum up to 1. It is suitable for tasks such as image recognition, natural language processing, or sentiment analysis with multiple classes.

1. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?

Loss of Generalization

Oscillations or Instability

Slower Convergence or Ineffective Learning

Difficulty Escaping Local Minima

Inefficiency

1. Name three ways you can produce a sparse model.

Producing sparse models refers to reducing the number of non-zero parameters or activations in a model, which can lead to several benefits such as improved efficiency, interpretability, and generalization. Here are three ways to produce sparse models:

L1 Regularization (Lasso):

Dropout:

Pruning:

1. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout?

Dropout can affect the training process by introducing additional computational overhead, but its impact on training speed depends on various factors. Here are the effects of dropout on training and inference:

Training Speed: Dropout can potentially slow down the training process since it requires additional computations.

Inference Speed: Dropout does not affect inference speed significantly. During inference, dropout is typically turned off or scaled down, and the full network is used without dropout.

MC Dropout: MC Dropout (Monte Carlo Dropout) is a technique that extends dropout to the inference phase.

Overall, dropout may marginally slow down training due to the additional computations, but it does not significantly impact inference speed. MC Dropout extends dropout to inference, introducing some additional computational cost, but it can provide more accurate predictions and uncertainty estimation.

1. Practice training a deep neural network on the CIFAR10 image dataset:
   1. 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.

import tensorflow as tf

# Define the DNN architecture

num\_hidden\_layers = 20

num\_neurons = 100

input\_shape = [None, input\_size] # Adjust input\_size based on your input data

model = tf.keras.models.Sequential()

# Add the input layer

model.add(tf.keras.layers.InputLayer(input\_shape=input\_shape))

# Add the hidden layers

for \_ in range(num\_hidden\_layers):

model.add(tf.keras.layers.Dense(

num\_neurons,

kernel\_initializer=tf.keras.initializers.HeNormal(),

activation=tf.keras.activations.elu

))

# Add the output layer

model.add(tf.keras.layers.Dense(output\_size, activation='softmax')) # Adjust output\_size based on your task

# Compile the model

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Print the summary of the model

model.summary()

* 1. 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.

import tensorflow as tf

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.callbacks import EarlyStopping

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Normalize pixel values to the range [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Convert class labels to one-hot encoded vectors

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes=10)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes=10)

# Define the DNN architecture

model = Sequential()

# Add the input layer

model.add(Dense(num\_neurons, activation='elu', input\_shape=(32, 32, 3)))

# Add the hidden layers

for \_ in range(num\_hidden\_layers):

model.add(Dense(num\_neurons, activation='elu'))

# Add the output layer

model.add(Dense(10, activation='softmax'))

# Compile the model with Nadam optimizer

optimizer = Nadam(learning\_rate=0.001)

model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])

# Define early stopping criteria

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, verbose=1)

# Train the model with early stopping

history = model.fit(x\_train, y\_train, batch\_size=128, epochs=100, validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping])

# Evaluate the model on test data

\_, accuracy = model.evaluate(x\_test, y\_test)

print('Test accuracy:', accuracy)

* 1. 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?

import tensorflow as tf

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, BatchNormalization

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Normalize pixel values to the range [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Convert class labels to one-hot encoded vectors

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes=10)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes=10)

# Define the DNN architecture without batch normalization

model\_without\_bn = Sequential()

model\_without\_bn.add(Dense(num\_neurons, activation='elu', input\_shape=(32, 32, 3)))

for \_ in range(num\_hidden\_layers):

model\_without\_bn.add(Dense(num\_neurons, activation='elu'))

model\_without\_bn.add(Dense(10, activation='softmax'))

model\_without\_bn.compile(optimizer=Nadam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy'])

# Define the DNN architecture with batch normalization

model\_with\_bn = Sequential()

model\_with\_bn.add(Dense(num\_neurons, activation='elu', input\_shape=(32, 32, 3)))

model\_with\_bn.add(BatchNormalization())

for \_ in range(num\_hidden\_layers):

model\_with\_bn.add(Dense(num\_neurons, activation='elu'))

model\_with\_bn.add(BatchNormalization())

model\_with\_bn.add(Dense(10, activation='softmax'))

model\_with\_bn.compile(optimizer=Nadam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy'])

# Define early stopping criteria

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, verbose=1)

# Train the models with and without batch normalization

history\_without\_bn = model\_without\_bn.fit(x\_train, y\_train, batch\_size=128, epochs=100, validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping], verbose=1)

history\_with\_bn = model\_with\_bn.fit(x\_train, y\_train, batch\_size=128, epochs=100, validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping], verbose=1)

# Plot the learning curves

plt.plot(history\_without\_bn.history['loss'], label='Without Batch Normalization - Training')

plt.plot(history\_without\_bn.history['val\_loss'], label='Without Batch Normalization - Validation')

plt.plot(history\_with\_bn.history['loss'], label='With Batch Normalization - Training')

plt.plot(history\_with\_bn.history['val\_loss'], label='With Batch Normalization - Validation')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Learning Curves')

plt.legend()

plt.show()

# Evaluate the models on test data

\_, accuracy\_without\_bn = model\_without\_bn.evaluate(x\_test, y\_test)

\_, accuracy\_with\_bn = model\_with\_bn.evaluate(x\_test, y\_test)

print('Test accuracy without Batch Normalization:', accuracy\_without\_bn)

print('Test accuracy with Batch Normalization:', accuracy\_with\_bn)

* 1. Try replacing Batch Normalization with SELU, and make the necessary adjustements 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.).

import tensorflow as tf

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.initializers import lecun\_normal

from tensorflow.keras.activations import selu

from tensorflow.keras.callbacks import EarlyStopping

import numpy as np

# Load CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Convert pixel values to float32

x\_train = x\_train.astype('float32')

x\_test = x\_test.astype('float32')

# Standardize the input features

x\_train\_mean = np.mean(x\_train, axis=(0, 1, 2))

x\_train\_std = np.std(x\_train, axis=(0, 1, 2))

x\_train = (x\_train - x\_train\_mean) / x\_train\_std

x\_test = (x\_test - x\_train\_mean) / x\_train\_std

# Convert class labels to one-hot encoded vectors

num\_classes = 10

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

# Define the DNN architecture with SELU activation

model = Sequential()

# Add the input layer with LeCun normal initialization

model.add(Dense(num\_neurons, input\_shape=(32, 32, 3), kernel\_initializer=lecun\_normal(), activation=selu))

# Add the hidden layers with SELU activation

for \_ in range(num\_hidden\_layers):

model.add(Dense(num\_neurons, kernel\_initializer=lecun\_normal(), activation=selu))

# Add the output layer

model.add(Dense(num\_classes, activation='softmax'))

# Compile the model with Nadam optimizer

optimizer = Nadam(learning\_rate=0.001)

model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])

# Define early stopping criteria

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, verbose=1)

# Train the model with early stopping

history = model.fit(x\_train, y\_train, batch\_size=128, epochs=100, validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping])

# Evaluate the model on test data

\_, accuracy = model.evaluate(x\_test, y\_test)

print('Test accuracy:', accuracy)

* 1. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.

# Evaluate the model with AlphaDropout on test data

\_, accuracy\_with\_alpha\_dropout = model\_with\_alpha\_dropout.evaluate(x\_test, y\_test)

print('Test accuracy with AlphaDropout:', accuracy\_with\_alpha\_dropout)

# Enable MC Dropout during inference

mc\_iterations = 100 # Number of forward passes for MC Dropout

predictions = np.zeros((mc\_iterations, x\_test.shape[0], num\_classes))

for i in range(mc\_iterations):

predictions[i] = model\_with\_alpha\_dropout.predict(x\_test, batch\_size=128, verbose=1)

# Compute ensemble predictions and accuracy

ensemble\_predictions = np.mean(predictions, axis=0)

ensemble\_accuracy = np.mean(np.argmax(ensemble\_predictions, axis=1) == np.argmax(y\_test, axis=1))

print('Ensemble accuracy with MC Dropout:', ensemble\_accuracy)