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

🡪 Initializing all weights to the same value, even if it's a random value selected using He initialization, is not generally recommended. He initialization is a technique that aims to initialize the weights in a way that the variance of the outputs of each neuron remains constant across the layers. It takes into account the number of input units to a neuron and scales the initial weights accordingly. The idea is to prevent the signal from vanishing or exploding as it propagates through the network during training.

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

🡪 Initializing the bias terms to 0 is a common practice and generally considered acceptable in neural network initialization. Setting the bias terms to 0 does not introduce any issues and is often the default initialization approach. Bias terms are used to introduce an offset in each neuron's activation, allowing the network to learn and model data patterns that may not be centred around zero. By setting the biases to 0 initially, you are effectively assuming that the data is centred around zero, which is a reasonable assumption in many cases.

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

🡪 The SELU (Scaled Exponential Linear Unit) activation function offers several advantages over the ReLU (Rectified Linear Unit) activation function.

1. Self-normalization: One significant advantage of SELU over ReLU is its self-normalizing property. The SELU activation function is designed to preserve the mean and variance of the inputs during forward propagation, helping to stabilize and regularize the learning process. This can alleviate the vanishing gradient problem and enable deep neural networks to train more effectively, especially in deep architectures with many layers.

2. Continuous and smooth: Unlike ReLU, which is piecewise linear and has a discontinuity at zero, SELU is a smooth and continuous activation function. The smoothness of SELU can be beneficial in certain scenarios, as it allows for gradient propagation and smoother optimization. It can help mitigate gradient-related issues like dead neurons and gradient explosions, promoting more stable training.

3. Improved learning dynamics: SELU has been shown to exhibit improved learning dynamics compared to ReLU. The self-normalizing property and smoothness of SELU contribute to better gradient flow and convergence characteristics. Neural networks utilizing SELU activations have demonstrated improved training speed, better generalization performance, and higher accuracy on various tasks, especially when dealing with complex and deep architectures.

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 have their strengths and are suitable for different scenarios.

1. SELU (Scaled Exponential Linear Unit):

- Use SELU when working with deep neural networks, especially in architectures with many layers. SELU's self-normalizing property helps mitigate the vanishing gradient problem and enables more stable and effective training.

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

- Use leaky ReLU variants when you want to mitigate the "dying ReLU" problem, where neurons become non-responsive (output zero) and stop learning. Leaky ReLU allows a small gradient for negative inputs, preventing complete "dead" neurons and encouraging better gradient flow.

3. ReLU (Rectified Linear Unit):

- ReLU is widely used as a default choice due to its simplicity and computational efficiency.

- Use ReLU when you want to introduce non-linearity into your model and benefit from sparsity (as ReLU sets negative values to zero).

- It is often a good choice for hidden layers in deep neural networks, where its effectiveness has been demonstrated in many applications.

4. tanh (Hyperbolic Tangent):

- tanh is commonly used in recurrent neural networks (RNNs) and as an alternative to sigmoid activation.

- Use tanh when you need an activation function that squashes input values between -1 and 1, allowing negative and positive values.

- It can be useful in models that require inputs in a bounded range.

5. logistic (Sigmoid):

- logistic activation is commonly used in binary classification problems.

- Use logistic when you need to model a probability-like output, where values range from 0 to 1.

- It is often used as the final activation function in binary classification tasks.

6. softmax:

- softmax activation is typically used in multi-class classification tasks.

- Use softmax when you need to assign probabilities to multiple classes, ensuring that the output probabilities sum up to 1.

- It is often used as the final activation function in the output layer of neural networks for multi-class classification.

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

🡪 When setting the momentum hyperparameter too close to 1 in an SGD optimizer, several issues may arise:

1. Overshooting and instability: Momentum in SGD helps accelerate the learning process by accumulating previous gradients and adding a fraction of it to the current update step. When the momentum is set too close to 1, the accumulated gradient becomes excessively large, leading to overshooting and instability during training. The gradient updates may oscillate or diverge, making it difficult for the model to converge to a good solution.

2. Slower convergence: While higher momentum can help SGD converge faster in some cases, setting it extremely close to 1 can actually have the opposite effect. The excessive momentum can cause the learning process to become less efficient, slowing down convergence or preventing the model from converging at all. The updates may become erratic, resulting in a longer training time and potentially hindering the model's performance.

3. Difficulty in escaping local minima: Higher momentum allows the optimizer to overcome shallow local minima and move towards flatter regions of the loss landscape. However, setting the momentum too close to 1 may make it difficult for the optimizer to escape deep local minima or saddle points. The excessive momentum can cause the optimizer to overshoot or get trapped in undesirable regions, preventing the model from finding better solutions.

4. Sensitivity to learning rate: The choice of learning rate plays a crucial role in conjunction with the momentum hyperparameter. Setting the momentum too close to 1 amplifies the influence of the learning rate on the updates. If the learning rate is too large, it can magnify the negative effects mentioned above. Conversely, if the learning rate is too small, it may cause slow convergence or stalling.

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

🡪 1. L1 Regularization (Lasso Regression): Applying L1 regularization to a model involves adding a penalty term to the loss function proportional to the sum of the absolute values of the model's weights. By optimizing the loss function with L1 regularization, the model tends to drive some weights to exactly zero, effectively creating sparsity. L1 regularization promotes feature selection and can be especially useful when dealing with high-dimensional data.

2. Dropout: Dropout is a regularization technique commonly used in neural networks to reduce overfitting. During training, dropout randomly sets a fraction of the neuron activations (outputs) to zero at each update, effectively "dropping out" those neurons. This introduces noise and prevents neurons from relying too heavily on specific inputs or co-adapting, encouraging the network to learn more robust and sparse representations.

3. Pruning: Pruning involves removing or setting a subset of weights in a trained model to zero based on their magnitudes or other criteria. This process can be done during or after training. Weight pruning techniques can identify and eliminate less important connections, resulting in a sparse model with reduced computational requirements. Pruning can be applied globally across the entire model or locally within specific layers or neurons.

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

🡪 Dropout can indeed slightly slow down the training process compared to a model without dropout. During training, dropout introduces randomness by zeroing out a fraction of neuron activations, which effectively creates an ensemble of thinned networks. As a result, the forward and backward passes in each training iteration involve more computations due to the stochastic nature of dropout.

However, dropout can still be a valuable regularization technique despite the slight increase in training time. By preventing overfitting and improving generalization, dropout can lead to better performance on unseen data, which outweighs the marginal increase in training time.

In terms of inference, dropout does not slow down the process of making predictions on new instances. During inference, dropout is typically turned off or scaled down to preserve the expected activations of neurons. As a result, there is no random dropout applied during inference, and the model behaves as if all neurons are active. Therefore, inference with dropout does not introduce any additional computation overhead compared to a model without dropout.

MC Dropout (Monte Carlo Dropout) is an extension of traditional dropout that leverages the dropout technique during inference to estimate model uncertainty. Instead of simply turning off dropout during inference, MC Dropout involves performing multiple forward passes with dropout enabled, resulting in multiple predictions per input. By averaging or analyzing the distribution of these predictions, MC Dropout provides a measure of uncertainty or confidence in the model's predictions. While MC Dropout involves performing multiple forward passes, the additional computational overhead is generally acceptable, especially for small to medium-sized networks.

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

from tensorflow import keras

# Load the CIFAR10 dataset

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

# Normalize the data

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

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

# Build the model

model = keras.Sequential()

model.add(keras.layers.Flatten(input\_shape=(32, 32, 3)))

for \_ in range(20):

model.add(keras.layers.Dense(100, kernel\_initializer='he\_normal', activation='elu'))

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

# Compile the model

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

# Train the model

history = model.fit(x\_train, y\_train, epochs=30, validation\_data=(x\_test, y\_test))

* 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 import keras

from tensorflow.keras import layers

from tensorflow.keras.datasets import cifar10

# Load the CIFAR10 dataset

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

# Convert pixel values to float and normalize

x\_train = x\_train.astype("float32") / 255

x\_test = x\_test.astype("float32") / 255

# Convert labels to one-hot encoding

num\_classes = 10

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

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

# Define the neural network architecture

model = keras.Sequential(

[

layers.Conv2D(32, (3, 3), activation="relu", input\_shape=(32, 32, 3)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation="relu"),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(64, activation="relu"),

layers.Dense(num\_classes, activation="softmax"),

]

)

# Compile the model

optimizer = keras.optimizers.Nadam()

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

# Set up early stopping

early\_stopping = keras.callbacks.EarlyStopping(

monitor="val\_loss", patience=3, restore\_best\_weights=True

)

# Train the model

history = model.fit(

x\_train,

y\_train,

epochs=50,

batch\_size=32,

validation\_split=0.1,

callbacks=[early\_stopping],

)

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=0)

print("Test accuracy:", test\_acc)

* 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 import keras

from tensorflow.keras import layers

from tensorflow.keras.datasets import cifar10

# Load the CIFAR10 dataset

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

# Convert pixel values to float and normalize

x\_train = x\_train.astype("float32") / 255

x\_test = x\_test.astype("float32") / 255

# Convert labels to one-hot encoding

num\_classes = 10

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

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

# Define the neural network architecture with Batch Normalization

model = keras.Sequential(

[

layers.Conv2D(32, (3, 3), activation="relu", input\_shape=(32, 32, 3)),

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation="relu"),

layers.BatchNormalization(),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(64, activation="relu"),

layers.BatchNormalization(),

layers.Dense(num\_classes, activation="softmax"),

]

)

# Compile the model

optimizer = keras.optimizers.Nadam()

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

# Set up early stopping

early\_stopping = keras.callbacks.EarlyStopping(

monitor="val\_loss", patience=3, restore\_best\_weights=True

)

# Train the model

history\_bn = model.fit(

x\_train,

y\_train,

epochs=50,

batch\_size=32,

validation\_split=0.1,

callbacks=[early\_stopping],

)

# Evaluate the model on the test set

test\_loss\_bn, test\_acc\_bn = model.evaluate(x\_test, y\_test, verbose=0)

print("Test accuracy with Batch Normalization:", test\_acc\_bn)

import matplotlib.pyplot as plt

# Plot the learning curves

plt.plot(history.history["val\_accuracy"], label="Without Batch Normalization")

plt.plot(history\_bn.history["val\_accuracy"], label="With Batch Normalization")

plt.title("Validation Accuracy")

plt.xlabel("Epoch")

plt.ylabel("Accuracy")

plt.legend()

plt.show()

* 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.).

🡪 from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten, Dense, Dropout, Conv2D, MaxPooling2D

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.initializers import lecun\_normal

from tensorflow.keras.utils import normalize

from tensorflow.keras.layers import Activation

# Load the CIFAR10 dataset

(X\_train, y\_train), (X\_test, y\_test) = keras.datasets.cifar10.load\_data()

# Normalize the input data

X\_train = normalize(X\_train, axis=1)

X\_test = normalize(X\_test, axis=1)

# Define the model architecture

model = Sequential([

Conv2D(32, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same', input\_shape=(32,32,3)),

Conv2D(32, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

MaxPooling2D(pool\_size=(2,2)),

Dropout(0.25),

Conv2D(64, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

Conv2D(64, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

MaxPooling2D(pool\_size=(2,2)),

Dropout(0.25),

Flatten(),

Dense(512, activation='selu', kernel\_initializer=lecun\_normal()),

Dropout(0.5),

Dense(10, activation='softmax')

])

# Compile the model with Nadam optimizer

optimizer = Nadam(lr=0.001)

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

# Define early stopping

early\_stopping = EarlyStopping(patience=10, restore\_best\_weights=True)

# Train the model with early stopping

history = model.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_split=0.2, callbacks=[early\_stopping])

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print('Test accuracy with SELU:', test\_acc)

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

🡪 from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten, Dense, Dropout, Conv2D, MaxPooling2D, AlphaDropout

from tensorflow.keras.optimizers import Nadam

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.initializers import lecun\_normal

from tensorflow.keras.utils import normalize

from tensorflow.keras.layers import Activation

import numpy as np

# Load the CIFAR10 dataset

(X\_train, y\_train), (X\_test, y\_test) = keras.datasets.cifar10.load\_data()

# Normalize the input data

X\_train = normalize(X\_train, axis=1)

X\_test = normalize(X\_test, axis=1)

# Define the model architecture with alpha dropout

model = Sequential([

Conv2D(32, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same', input\_shape=(32,32,3)),

Conv2D(32, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

MaxPooling2D(pool\_size=(2,2)),

AlphaDropout(0.1),

Conv2D(64, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

Conv2D(64, (3,3), activation='selu', kernel\_initializer=lecun\_normal(), padding='same'),

MaxPooling2D(pool\_size=(2,2)),

AlphaDropout(0.1),

Flatten(),

Dense(512, activation='selu', kernel\_initializer=lecun\_normal()),

AlphaDropout(0.5),

Dense(10, activation='softmax')

])

# Compile the model with Nadam optimizer

optimizer = Nadam(lr=0.001)

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

# Define early stopping

early\_stopping = EarlyStopping(patience=10, restore\_best\_weights=True)

# Train the model with alpha dropout and early stopping

history = model.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_split=0.2, callbacks=[early\_stopping])

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print('Test accuracy with alpha dropout:', test\_acc)

# Use MC Dropout for improved accuracy without retraining the model

n\_samples = 100

y\_probs = np.stack([model.predict(X\_test, batch\_size=32, verbose=1) for \_ in range(n\_samples)])

y\_mean = y\_probs.mean(axis=0)

y\_std = y\_probs.std(axis=0)

y\_pred = np.argmax(y\_mean, axis=1)

test\_acc\_mc = (y\_pred == y\_test.squeeze()).mean()

print('Test accuracy with MC Dropout:', test\_acc\_mc)