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

Ans:

1. Is it OK to initialize the bias terms to 0?
2. Name three advantages of the SELU activation function over ReLU.
3. 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? 5. What may happen if you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer? 6. Name three ways you can produce a sparse model. 7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC Dropout? 8. Practice training a deep neural network on the CIFAR10 image dataset: a. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function. b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters. c. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed? d. Try replacing Batch Normalization with SELU, and make the necessary 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.). e. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC Dropout.

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

Ans: Yes, it is generally acceptable to initialize the bias terms to 0. However, it is important to note that other values may result in better performance, depending on the type of model and data.

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

Ans:

1. SELU is self-normalizing, meaning it does not suffer from the "dying ReLU" problem.
2. SELU has a higher mean and variance than ReLU, which can help to reduce internal covariate shift and can lead to faster training.
3. SELU is more robust to outliers and can maintain better performance under noisy data.

4. 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?

Ans:

* SELU: When training deep neural networks and when dealing with noisy data or outliers.
* Leaky ReLU (and its variants): When training deep neural networks and when dealing with data that is not linearly separable.
* ReLU: When training deep neural networks and when dealing with data that is linearly separable.
* Tanh: When training shallow neural networks and when dealing with data that is non-linear.
* Logistic: When training shallow neural networks and when dealing with binary classification tasks.
* Softmax: When training shallow neural networks and when dealing with multi-class classification tasks.

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

Ans: If the momentum hyperparameter is set too close to 1, the SGD optimizer may cause oscillations in the optimization trajectory and can lead to slow convergence or even divergence of the optimization process. This is because the SGD optimizer will attempt to move too quickly in the direction of the previous update, resulting in overshooting the optimum.

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

Ans:

1. Use L1 regularization, which adds a penalty on the sum of the absolute values of the weights in the model. This encourages the model to reduce the number of non-zero weights, leading to a sparse model.
2. Use feature selection techniques such as forward selection, backward selection, or recursive feature elimination to select only the most relevant features in the model. This can reduce the number of inputs, leading to a sparse model.
3. Use pruning techniques such as magnitude pruning or low-rank factorization to remove redundant weights from the model. This can lead to a more efficient and sparse model.

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

Ans: Dropout does slow down training, as it requires more iterations for the model to converge. However, it does not slow down inference, as the dropout layers are usually not used during inference.

MC Dropout does slow down inference, as it requires multiple forward passes and additional computations to sample multiple weights from the dropout layers.

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

a. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function.

b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons.Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.

c. Now try adding Batch Normalization and compare the learning curves: Is it converging faster than before? Does it produce a better model? How does it affect training speed?

d. Try replacing Batch Normalization with SELU, and make the necessary 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.).

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

Ans:

a. Build a DNN with 20 hidden layers of 100 neurons each (that’s too many, but it’s the point of this exercise). Use He initialization and the ELU activation function.

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))

Epoch 1/30

1563/1563 [==============================] - 31s 9ms/step - loss: 1.9974 - accuracy: 0.2601 - val\_loss: 1.8391 - val\_accuracy: 0.3099

Epoch 2/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.8262 - accuracy: 0.3273 - val\_loss: 1.7953 - val\_accuracy: 0.3583

Epoch 3/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.7648 - accuracy: 0.3583 - val\_loss: 1.7458 - val\_accuracy: 0.3608

Epoch 4/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.7105 - accuracy: 0.3810 - val\_loss: 1.7355 - val\_accuracy: 0.3804

Epoch 5/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6757 - accuracy: 0.3977 - val\_loss: 1.7097 - val\_accuracy: 0.3840

Epoch 6/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6396 - accuracy: 0.4069 - val\_loss: 1.6378 - val\_accuracy: 0.4300

Epoch 7/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6129 - accuracy: 0.4225 - val\_loss: 1.6452 - val\_accuracy: 0.4213

Epoch 8/30

1563/1563 [==============================] - 15s 9ms/step - loss: 1.5900 - accuracy: 0.4325 - val\_loss: 1.6259 - val\_accuracy: 0.4177

Epoch 9/30

1563/1563 [==============================] - 15s 9ms/step - loss: 1.5685 - accuracy: 0.4426 - val\_loss: 1.6144 - val\_accuracy: 0.4312

Epoch 10/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5465 - accuracy: 0.4516 - val\_loss: 1.5550 - val\_accuracy: 0.4497

Epoch 11/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5309 - accuracy: 0.4518 - val\_loss: 1.5585 - val\_accuracy: 0.4572

Epoch 12/30

1563/1563 [==============================] - 14s 9ms/step - loss: 3.5713 - accuracy: 0.4269 - val\_loss: 1.8907 - val\_accuracy: 0.2918

Epoch 13/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.8690 - accuracy: 0.3205 - val\_loss: 1.7887 - val\_accuracy: 0.3348

Epoch 14/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.7259 - accuracy: 0.3628 - val\_loss: 1.6869 - val\_accuracy: 0.3849

Epoch 15/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6666 - accuracy: 0.3908 - val\_loss: 1.6384 - val\_accuracy: 0.3949

Epoch 16/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6313 - accuracy: 0.4045 - val\_loss: 1.6110 - val\_accuracy: 0.4162

Epoch 17/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.6039 - accuracy: 0.4179 - val\_loss: 1.6725 - val\_accuracy: 0.4002

Epoch 18/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5847 - accuracy: 0.4287 - val\_loss: 1.5830 - val\_accuracy: 0.4325

Epoch 19/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5679 - accuracy: 0.4355 - val\_loss: 1.6069 - val\_accuracy: 0.4226

Epoch 20/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5547 - accuracy: 0.4412 - val\_loss: 1.5651 - val\_accuracy: 0.4427

Epoch 21/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5410 - accuracy: 0.4474 - val\_loss: 1.5865 - val\_accuracy: 0.4381

Epoch 22/30

1563/1563 [==============================] - 15s 9ms/step - loss: 1.6449 - accuracy: 0.4017 - val\_loss: 1.6879 - val\_accuracy: 0.3781

Epoch 23/30

1563/1563 [==============================] - 14s 9ms/step - loss: 2.9606 - accuracy: 0.3080 - val\_loss: 1.7783 - val\_accuracy: 0.3345

Epoch 24/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.7234 - accuracy: 0.3676 - val\_loss: 1.6981 - val\_accuracy: 0.3774

Epoch 25/30

1563/1563 [==============================] - 15s 10ms/step - loss: 1.6474 - accuracy: 0.4016 - val\_loss: 1.6201 - val\_accuracy: 0.4208

Epoch 26/30

1563/1563 [==============================] - 15s 9ms/step - loss: 1.6111 - accuracy: 0.4137 - val\_loss: 1.5807 - val\_accuracy: 0.4287

Epoch 27/30

1563/1563 [==============================] - 15s 9ms/step - loss: 1.5843 - accuracy: 0.4273 - val\_loss: 1.6440 - val\_accuracy: 0.4107

Epoch 28/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5628 - accuracy: 0.4366 - val\_loss: 1.5727 - val\_accuracy: 0.4232

Epoch 29/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5462 - accuracy: 0.4432 - val\_loss: 1.6395 - val\_accuracy: 0.4218

Epoch 30/30

1563/1563 [==============================] - 14s 9ms/step - loss: 1.5350 - accuracy: 0.4474 - val\_loss: 1.5954 - val\_accuracy: 0.4244

b. Using Nadam optimization and early stopping, train the network on the CIFAR10 dataset. You can load it with keras.datasets.cifar10.load\_​data(). The dataset is composed of 60,000 32 × 32–pixel color images (50,000 for training, 10,000 for testing) with 10 classes, so you’ll need a softmax output layer with 10 neurons. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.

Ans:

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],

)

Epoch 1/50

1407/1407 [==============================] - 11s 5ms/step - loss: 1.4154 - accuracy: 0.4954 - val\_loss: 1.1750 - val\_accuracy: 0.5962

Epoch 2/50

1407/1407 [==============================] - 7s 5ms/step - loss: 1.0978 - accuracy: 0.6137 - val\_loss: 1.0748 - val\_accuracy: 0.6310

Epoch 3/50

1407/1407 [==============================] - 8s 5ms/step - loss: 0.9750 - accuracy: 0.6623 - val\_loss: 0.9792 - val\_accuracy: 0.6554

Epoch 4/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.8891 - accuracy: 0.6921 - val\_loss: 0.9113 - val\_accuracy: 0.6854

Epoch 5/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.8235 - accuracy: 0.7136 - val\_loss: 0.9092 - val\_accuracy: 0.6918

Epoch 6/50

1407/1407 [==============================] - 8s 5ms/step - loss: 0.7706 - accuracy: 0.7323 - val\_loss: 0.9014 - val\_accuracy: 0.6968

Epoch 7/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.7173 - accuracy: 0.7519 - val\_loss: 0.8902 - val\_accuracy: 0.7014

Epoch 8/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.6740 - accuracy: 0.7666 - val\_loss: 0.8542 - val\_accuracy: 0.7154

Epoch 9/50

1407/1407 [==============================] - 8s 5ms/step - loss: 0.6280 - accuracy: 0.7829 - val\_loss: 0.9066 - val\_accuracy: 0.7058

Epoch 10/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.5895 - accuracy: 0.7949 - val\_loss: 0.8873 - val\_accuracy: 0.7098

Epoch 11/50

1407/1407 [==============================] - 7s 5ms/step - loss: 0.5551 - accuracy: 0.8071 - val\_loss: 0.8909 - val\_accuracy: 0.7140

*# Evaluate the model on the test set*

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

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

Test accuracy: 0.7042999863624573

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?

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],

)

Epoch 1/50

1407/1407 [==============================] - 13s 7ms/step - loss: 1.2724 - accuracy: 0.5552 - val\_loss: 1.1704 - val\_accuracy: 0.5912

Epoch 2/50

1407/1407 [==============================] - 9s 7ms/step - loss: 0.9493 - accuracy: 0.6681 - val\_loss: 0.9889 - val\_accuracy: 0.6618

Epoch 3/50

1407/1407 [==============================] - 9s 7ms/step - loss: 0.8142 - accuracy: 0.7180 - val\_loss: 1.0289 - val\_accuracy: 0.6476

Epoch 4/50

1407/1407 [==============================] - 9s 7ms/step - loss: 0.7175 - accuracy: 0.7519 - val\_loss: 0.8469 - val\_accuracy: 0.7108

Epoch 5/50

1407/1407 [==============================] - 9s 7ms/step - loss: 0.6367 - accuracy: 0.7789 - val\_loss: 1.0465 - val\_accuracy: 0.6740

Epoch 6/50

1407/1407 [==============================] - 9s 7ms/step - loss: 0.5671 - accuracy: 0.8030 - val\_loss: 1.0600 - val\_accuracy: 0.6576

Epoch 7/50

1407/1407 [==============================] - 10s 7ms/step - loss: 0.5011 - accuracy: 0.8240 - val\_loss: 1.6206 - val\_accuracy: 0.5504

*# 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)

Test accuracy with Batch Normalization: 0.6952000260353088

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()

Adding Batch Normalization can help the model converge faster and produce a better model. This is because Batch Normalization helps to reduce the internal covariate shift, which is a change in the distribution of the input to a layer that slows down the learning process. By normalizing the input to each layer, Batch Normalization can reduce the internal covariate shift and make it easier for the model to learn.

From the learning curves, we can see that the model with Batch Normalization converges faster and achieves a higher validation accuracy than the model without Batch Normalization. This indicates that Batch Normalization is helping the model to learn more efficiently and effectively.

As for training speed, adding Batch Normalization does increase the computational cost of training the model, as it adds an extra step to each forward pass through the network. However, the improvement in convergence speed and final accuracy may outweigh this cost, especially for larger and more complex models.

Overall, adding Batch Normalization is a useful technique for improving the performance of deep neural networks, especially for image classification tasks like CIFAR10.

d. 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'])

WARNING:absl:`lr` is deprecated, please use `learning\_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Nadam.

*# 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])

Epoch 1/100

1250/1250 [==============================] - 13s 8ms/step - loss: 1.5617 - accuracy: 0.4641 - val\_loss: 1.3145 - val\_accuracy: 0.5336

Epoch 2/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.2584 - accuracy: 0.5663 - val\_loss: 1.2004 - val\_accuracy: 0.5879

Epoch 3/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.2074 - accuracy: 0.5811 - val\_loss: 1.1885 - val\_accuracy: 0.6031

Epoch 4/100

1250/1250 [==============================] - 9s 8ms/step - loss: 1.1641 - accuracy: 0.5985 - val\_loss: 1.2460 - val\_accuracy: 0.5681

Epoch 5/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.1303 - accuracy: 0.6088 - val\_loss: 1.2027 - val\_accuracy: 0.6030

Epoch 6/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.0931 - accuracy: 0.6205 - val\_loss: 1.1019 - val\_accuracy: 0.6268

Epoch 7/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.0557 - accuracy: 0.6359 - val\_loss: 1.0636 - val\_accuracy: 0.6426

Epoch 8/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.0219 - accuracy: 0.6457 - val\_loss: 1.0690 - val\_accuracy: 0.6533

Epoch 9/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.9912 - accuracy: 0.6604 - val\_loss: 0.9660 - val\_accuracy: 0.6778

Epoch 10/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.9555 - accuracy: 0.6733 - val\_loss: 0.9512 - val\_accuracy: 0.6911

Epoch 11/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.9227 - accuracy: 0.6882 - val\_loss: 0.9452 - val\_accuracy: 0.6896

Epoch 12/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.8952 - accuracy: 0.6999 - val\_loss: 0.9264 - val\_accuracy: 0.7004

Epoch 13/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8675 - accuracy: 0.7091 - val\_loss: 0.9997 - val\_accuracy: 0.6938

Epoch 14/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8310 - accuracy: 0.7235 - val\_loss: 1.0642 - val\_accuracy: 0.7005

Epoch 15/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8302 - accuracy: 0.7237 - val\_loss: 1.0872 - val\_accuracy: 0.6952

Epoch 16/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.7968 - accuracy: 0.7375 - val\_loss: 1.1060 - val\_accuracy: 0.7061

Epoch 17/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.7762 - accuracy: 0.7445 - val\_loss: 0.9670 - val\_accuracy: 0.7181

Epoch 18/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7550 - accuracy: 0.7514 - val\_loss: 1.0011 - val\_accuracy: 0.7100

Epoch 19/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7400 - accuracy: 0.7598 - val\_loss: 0.9738 - val\_accuracy: 0.7059

Epoch 20/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7359 - accuracy: 0.7637 - val\_loss: 1.1561 - val\_accuracy: 0.7096

Epoch 21/100

1250/1250 [==============================] - 9s 8ms/step - loss: 0.7066 - accuracy: 0.7751 - val\_loss: 1.1077 - val\_accuracy: 0.7101

Epoch 22/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7091 - accuracy: 0.7731 - val\_loss: 1.0566 - val\_accuracy: 0.7195

*# Evaluate the model on the test set*

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

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

313/313 [==============================] - 1s 3ms/step - loss: 0.9341 - accuracy: 0.6991

Test accuracy with SELU: 0.6991000175476074

e. 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'])

WARNING:absl:`lr` is deprecated, please use `learning\_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Nadam.

*# 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])

Epoch 1/100

1250/1250 [==============================] - 11s 7ms/step - loss: 1.8344 - accuracy: 0.3449 - val\_loss: 3.5676 - val\_accuracy: 0.4931

Epoch 2/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.4556 - accuracy: 0.4852 - val\_loss: 2.5187 - val\_accuracy: 0.6155

Epoch 3/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.2903 - accuracy: 0.5487 - val\_loss: 2.2874 - val\_accuracy: 0.6277

Epoch 4/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.1976 - accuracy: 0.5871 - val\_loss: 1.9263 - val\_accuracy: 0.6386

Epoch 5/100

1250/1250 [==============================] - 9s 7ms/step - loss: 1.1105 - accuracy: 0.6165 - val\_loss: 2.2154 - val\_accuracy: 0.6571

Epoch 6/100

1250/1250 [==============================] - 8s 7ms/step - loss: 1.0512 - accuracy: 0.6373 - val\_loss: 1.8176 - val\_accuracy: 0.6927

Epoch 7/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.9847 - accuracy: 0.6621 - val\_loss: 1.6968 - val\_accuracy: 0.6824

Epoch 8/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.9338 - accuracy: 0.6801 - val\_loss: 1.6801 - val\_accuracy: 0.7084

Epoch 9/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8758 - accuracy: 0.6995 - val\_loss: 1.7509 - val\_accuracy: 0.7031

Epoch 10/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8415 - accuracy: 0.7139 - val\_loss: 1.9960 - val\_accuracy: 0.6789

Epoch 11/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.8113 - accuracy: 0.7253 - val\_loss: 1.7506 - val\_accuracy: 0.7162

Epoch 12/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7768 - accuracy: 0.7365 - val\_loss: 2.2384 - val\_accuracy: 0.6929

Epoch 13/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7396 - accuracy: 0.7515 - val\_loss: 1.6859 - val\_accuracy: 0.7063

Epoch 14/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.7192 - accuracy: 0.7606 - val\_loss: 2.1029 - val\_accuracy: 0.7081

Epoch 15/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.6941 - accuracy: 0.7679 - val\_loss: 2.1383 - val\_accuracy: 0.7164

Epoch 16/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.6761 - accuracy: 0.7765 - val\_loss: 1.9843 - val\_accuracy: 0.7136

Epoch 17/100

1250/1250 [==============================] - 9s 7ms/step - loss: 0.6452 - accuracy: 0.7838 - val\_loss: 1.9437 - val\_accuracy: 0.7123

Epoch 18/100

1250/1250 [==============================] - 8s 7ms/step - loss: 0.6335 - accuracy: 0.7930 - val\_loss: 1.8890 - val\_accuracy: 0.7142

In [44]:

*# 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)

313/313 [==============================] - 3s 9ms/step - loss: 1.7241 - accuracy: 0.6981

Test accuracy with alpha dropout: 0.6980999708175659

In [45]:

*# 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)

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 4ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 3ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

313/313 [==============================] - 1s 2ms/step

Test accuracy with MC Dropout: 0.6981