**DEEP LEARNING ASSIGNMENT\_3**

**1. Is it OK to initialize all the weights to the same value as long as that value is selected**

**randomly using He initialization?**

When training a neural network, it is important to initialize the weights randomly to break the symmetry of the model. Symmetry in the initial weights implies that all neurons in a layer are essentially the same, so they will produce the same outputs, leading to identical gradients during backpropagation and reducing the model's ability to learn meaningful representations. This can result in slow convergence or even failure to converge.

Using a randomly selected weight initialization method like He initialization helps to overcome the problem of symmetry. However, initializing all weights to the same value, even if it is randomly selected, still has symmetry, which can still cause problems during training. It is recommended to initialize weights with a random distribution that follows the method recommended by the initialization technique to ensure that the weights are different and have the appropriate scale for the activation function used in the network.

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

Yes, it is common to initialize the bias terms to 0. This is because the bias term serves as a baseline offset for the neuron's output and does not participate in the weight updates during backpropagation. Initializing the bias terms to 0 allows the network to have a neutral starting point and make adjustments as needed during training. However, other non-zero initial values may also be used depending on the specific requirements of the problem.

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

hree advantages of the Scaled Exponential Linear Unit (SELU) activation function over the Rectified Linear Unit (ReLU) activation function are:

Self-Normalizing: SELU activation function has a self-normalizing property that ensures the outputs of the network remain in a narrow range and prevent vanishing or exploding gradients.

Network-wide convergence: SELU activation function guarantees network-wide convergence to a solution, while ReLU may face the dying ReLU problem in which neurons never activate again and thus become useless.

Increased accuracy: SELU activation function has been shown to improve the accuracy and stability of deep networks, especially in deep networks, compared to ReLU activation.

**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?**

The activation functions commonly used in different scenarios in neural network models are:

SELU: Scaled Exponential Linear Unit (SELU) is used in deep networks to improve stability and accuracy. It is often used in feedforward networks to avoid the vanishing and exploding gradient problem.

Leaky ReLU (and its variants): Leaky ReLU is used when the ReLU activation function causes the dying ReLU problem, in which some neurons never activate again and become useless. The leaky version allows a small, non-zero gradient when the input is negative, preventing the dead neurons problem.

ReLU: Rectified Linear Unit (ReLU) is a simple activation function that has proven to be effective in many cases, especially in computer vision and natural language processing tasks. It is fast to compute and does not have the vanishing gradient problem, making it a popular choice for feedforward networks.

tanh: Hyperbolic Tangent (tanh) is commonly used in recurrent neural networks (RNNs) and in situations where the network needs to model data with range (-1, 1). tanh has a smooth and continuous output range.

Logistic: Logistic or sigmoid activation function is used in binary classification problems, as it maps its input to a probability value between 0 and 1.

Softmax: Softmax activation function is used in multi-class classification problems, as it maps the inputs of a network to a probability distribution over multiple classes. Softmax is often used in the output layer of a neural network, with cross-entropy loss, to perform classification.

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

If the momentum hyperparameter is set too close to 1 (e.g., 0.99999) when using an Stochastic Gradient Descent (SGD) optimizer, the following problems may occur:

Oscillation: The gradient updates may oscillate instead of converge to a minimum, causing the optimizer to oscillate between different points in the loss landscape.

Slow convergence: Setting the momentum too close to 1 may cause the optimizer to move too quickly and overshoot the minimum, resulting in slow convergence or even divergence.

Getting stuck: In some cases, a high momentum value can cause the optimizer to get stuck in a poor local minimum, instead of reaching the global minimum.

In general, it is recommended to set the momentum hyperparameter to a value between 0.5 and 0.9 for best performance. A value close to 1 may work in some cases, but it is important to tune this hyperparameter carefully and monitor the optimization process to avoid the above problems.

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

Three ways to produce a sparse model are:

L1 Regularization: L1 regularization adds a penalty term to the loss function equal to the absolute value of the weights, causing many weights to become zero, producing a sparse model.

Pruning: Pruning involves removing the smallest or least important weights from a trained model, effectively making it sparse.

Structured Regularization: Structured regularization techniques, such as group Lasso or structured sparsity-inducing penalties, are designed to produce sparse models with a specific structure or pattern, such as grouping similar features together. These techniques encourage the model to have sparse and interpretable representations.

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

Dropout: Dropout is a regularization technique that randomly drops out neurons during training, reducing overfitting. It does slow down training, as it adds computational overhead during each forward and backward pass through the network. However, this slowdown is generally acceptable in most cases.

Inference: During inference, dropout is usually turned off, so it does not slow down predictions on new instances.

MC Dropout: Monte Carlo Dropout (MC Dropout) is a Bayesian variant of dropout that can be used to estimate model uncertainty. During inference, MC Dropout involves running the network multiple times with dropout turned on and averaging the predictions, which can be computationally expensive and slow down predictions. However, this added computational overhead is necessary to obtain reliable uncertainty estimates.

**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.**

Here is an example code in Python using the Keras deep learning library to build and train a deep neural network with 20 hidden layers of 100 neurons each on the CIFAR10 image dataset:

import numpy as np

import keras

from keras.datasets import cifar10

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation

from keras.optimizers import SGD

# Load the CIFAR10 data

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

# Preprocess the data

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

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

# Convert class vectors to binary class matrices

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

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

# Build the model

model = Sequential()

# Add hidden layers

for i in range(20):

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

# Add output layer

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

# Compile the model

model.compile(loss='categorical\_crossentropy',

optimizer=SGD(lr=0.01, momentum=0.9),

metrics=['accuracy'])

# Train the model

history = model.fit(x\_train, y\_train,

batch\_size=128,

epochs=100,

validation\_data=(x\_test, y\_test),

verbose=1)

This code builds a deep neural network with 20 hidden layers of 100 neurons each, using the Exponential Linear Unit (ELU) activation function and He normal initialization. The model is trained using Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.01 and momentum of 0.9. The model is trained for 100 epochs on the CIFAR10 dataset, and the accuracy is monitored on the validation set during training.

**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.**

Here is an example code in Python using the Keras deep learning library to train a deep neural network on the CIFAR10 image dataset using Nadam optimization and early stopping:

import numpy as np

import keras

from keras.datasets import cifar10

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation

from keras.optimizers import Nadam

from keras.callbacks import EarlyStopping

# Load the CIFAR10 data

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

# Preprocess the data

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

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

# Convert class vectors to binary class matrices

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

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

# Build the model

model = Sequential()

# Add hidden layers

for i in range(20):

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

# Add the output layer

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

# Compile the model

model.compile(loss='categorical\_crossentropy',

optimizer=Nadam(lr=0.002),

metrics=['accuracy'])

# Set up early stopping

early\_stopping = EarlyStopping(patience=10)

# Train the model

history = model.fit(x\_train, y\_train,

batch\_size=128,

epochs=100,

verbose=1,

validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping])

# Evaluate the model on the test data

score = model.evaluate(x\_test, y\_test, verbose=0)

print('Test loss:', score[0])

print('Test accuracy:', score[1])

**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.**