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

Ans. -No, all weights should be sampled independently; they should not all have the same initial value.

-One important goal of sampling weights randomly is to break symmetry: if all the weights have the same initial value, even if that value is not zero, then symmetry is not broken (i.e., all neurons in a given layer are equivalent), and backpropagation will be unable to break it. Concretely, this means that all the neurons in any given layer will always have the same weights. It’s like having just one neuron per layer, and much slower.

-It is virtually impossible for such a configuration to converge to a good solution.

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

Ans. It is perfectly fine to initialize the bias terms to zero. Some people like to initialize them just like weights, and that’s okay too; it does not make much difference.

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

Ans. 1. Similar to ReLUs, SELUs enable deep neural networks since there is no problem with vanishing gradients.

2. In contrast to ReLUs, SELUs cannot die.

3. SELUs on their own learn faster and better than other activation functions, even if they are combined with batch normalization.

**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. T-he SELU activation function is a good default.

-If you need the neural network to be as fast as possible, you can use one of the leaky ReLU variants instead (e.g., a simple leaky ReLU using the default hyperparameter value).

-simplicity of the ReLU activation function makes it many people’s preferred option, despite the fact that it is generally outperformed by SELU and leaky ReLU. However, the ReLU activation function’s ability to output precisely zero can be useful in some cases (e.g., see Chapter 17). Moreover, it can sometimes benefit from optimized implementation as well as from hardware acceleration.

-hyperbolic tangent (tanh) can be useful in the output layer if you need to output a number between –1 and 1, but nowadays it is not used much in hidden layers (except in recurrent nets).

-logistic activation function is also useful in the output layer when you need to estimate a probability (e.g., for binary classification),rare in hidden layer

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

If you set the momentum hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer, then the algorithm will likely pick up a lot of speed, hopefully moving roughly toward the global minimum, but its momentum will carry it right past the minimum. Then it will slow down and come back, accelerate again, overshoot again, and so on. It may oscillate this way many times before converging, so overall it will take much longer to converge than with a smaller momentum value.

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

-One way to produce a sparse model (i.e., with most weights equal to zero) is to train the model normally, then zero out tiny weights.

-For more sparsity, you can apply ℓ1 regularization during training, which pushes the optimizer toward sparsity.

-A third option is to use the TensorFlow Model Optimization Toolkit.

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

Yes, dropout does slow down training, in general roughly by a factor of two. However, it has no impact on inference speed since it is only turned on during training. MC Dropout is exactly like dropout during training, but it is still active during inference, so each inference is slowed down slightly. More importantly, when using MC Dropout you generally want to run inference 10 times or more to get better predictions. This means that making predictions is slowed down by a factor of 10 or more.

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

**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.**

works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.

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.

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?

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.).

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