**1. Is it Okay to Initialize All the Weights to the Same Value as Long as That Value is Selected Randomly Using He Initialization?**

No, it is generally **not okay** to initialize all the weights to the same value, even if it’s done randomly using He initialization or any other method. Here's why:

* **Symmetry Breaking**: If all weights are initialized the same, the neurons will behave identically during training. This means that each neuron will receive the same gradient updates, which will prevent the model from learning diverse representations.
* **He Initialization** is designed to **scale weights** based on the number of input units, providing proper variance to avoid vanishing or exploding gradients, but **each weight should be independently randomized** to break symmetry and allow each neuron to learn different features.

**2. Is it Okay to Initialize the Bias Terms to 0?**

Yes, it is generally **okay** to initialize bias terms to 0. In fact, this is a common practice. Here's why:

* **Bias Initialization**: The bias terms can be initialized to 0 without causing problems in most cases. Unlike weights, biases do not affect the symmetry of the network because they are added after the weighted sum of inputs. Initializing the bias to 0 allows the network to start with a neutral position, and during training, the optimizer will adjust the biases as necessary.
* **Avoiding Initial Symmetry**: The zero initialization of biases does not introduce any symmetry issue that might affect learning, unlike the weight initialization.

**3. Name Three Advantages of the ELU Activation Function Over ReLU.**

The **Exponential Linear Unit (ELU)** has some advantages over **ReLU**:

1. **No Dead Neurons**: ReLU can cause neurons to "die" if they fall into the negative part of the input space (producing zero output). ELU avoids this problem because it allows for negative outputs, ensuring that neurons can still learn even if their inputs are negative.
2. **Smoother Gradients**: ELU produces smoother gradients compared to ReLU, especially for negative values. This can result in faster and more stable convergence during training.
3. **Faster Learning**: ELU is designed to maintain the mean activations close to zero, which can reduce the shift in the distribution of the data as it flows through the layers, improving the optimization process and potentially leading to faster convergence.

**4. In Which Cases Would You Want to Use Each of the Following Activation Functions: ELU, Leaky ReLU (and its Variants), ReLU, Tanh, Logistic, and Softmax?**

* **ELU**: Use when you want to avoid dead neurons and need smoother gradients for faster and more stable learning. It's often used in deep networks where negative values are useful for learning.
* **Leaky ReLU (and its Variants)**: Use when you need to avoid the "dying ReLU" problem (where neurons become inactive). Leaky ReLU allows a small, non-zero gradient when the input is negative, preventing neurons from dying.
* **ReLU**: Use when you need a simple, efficient activation function that works well in many cases, particularly for networks with positive outputs. It is widely used in many architectures, especially convolutional networks.
* **Tanh**: Use when you need a **zero-centered** activation function that outputs values in the range of (-1, 1). This is suitable for problems where negative values are useful (e.g., in recurrent neural networks).
* **Logistic (Sigmoid)**: Use when you need an output between 0 and 1, particularly in binary classification tasks. It is commonly used in the output layer for binary classification problems or when you need probability values.
* **Softmax**: Use in the **output layer** for **multi-class classification** tasks. Softmax transforms a vector of values into a probability distribution across multiple classes (values between 0 and 1, summing to 1).

**5. What May Happen if You Set the Momentum Hyperparameter Too Close to 1 (e.g., 0.99999) When Using a MomentumOptimizer?**

If you set the **momentum** hyperparameter too close to 1, such as 0.99999, the optimizer will heavily "smooth" the updates over many steps, and this may result in:

* **Slow Convergence**: While momentum can help accelerate convergence in certain directions, a very high momentum value can make the optimizer "overshoot" and fail to adjust effectively to local minima, slowing down the learning process.
* **Difficulty in escaping saddle points**: High momentum can make the optimizer get stuck in saddle points or local minima, as it may not adjust the direction enough to escape these areas, causing slow or poor progress in training.

**6. Name Three Ways You Can Produce a Sparse Model.**

1. **L1 Regularization**: L1 regularization encourages sparsity by adding a penalty proportional to the absolute values of the weights, forcing some weights to become exactly zero, effectively "removing" those features.
2. **Dropout**: Dropout is a technique that randomly "drops" a proportion of neurons during training, which can lead to a sparse network by forcing the model to rely on fewer neurons at each step.
3. **Pruning**: Pruning involves removing (zeroing out) weights or neurons during or after training. This can result in a sparse model by eliminating less important weights or neurons that contribute little to the model's performance.

**7. Does Dropout Slow Down Training? Does It Slow Down Inference (i.e., Making Predictions on New Instances)?**

* **Training**: Yes, dropout **slows down training** because during training, a fraction of the neurons is randomly turned off at each step, so the network is effectively using a smaller subset of the neurons at each iteration. This leads to more training steps and longer overall training time.
* **Inference**: No, dropout **does not slow down inference**. During inference (prediction on new data), dropout is typically **turned off**, meaning all neurons are used. The model is run with all neurons active, and the weights are scaled appropriately based on the dropout rate to account for the neurons that were dropped during training.