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

Ans. In this case, each hidden unit will get exactly the same signal. E.g. if all weights are initialized to 1, each unit gets signal equal to sum of inputs (and outputs sigmoid(sum(inputs)) )

Is it okay to initialize the bias terms to 0?

Ans. It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights.

Name three advantages of the ELU activation function over ReLU.

Ans. It is difficult to generate nonlinearity (in the sense of “curvature”) with (piecewise) linearity (e.g. ReLU, Leaky ReLU). Hence it is difficult to effectively approximate smooth functions (e.g. polynomials) with a ReLU network.

On the other hand, it is difficult to generate linearity with nonlinearity. Try to approximate a piecewise linear function using a tanh network. It wouldn’t work very well, while it would be a child’s play for ReLUs.

ELU gives you the best of both worlds: it has nonlinearity and linearity at the same time. In addition, it doesn’t have the “dying ReLU” problem. The drawback is that it’s slightly slower to train.

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?

Ans. An advantage of ReLU is that you don’t have a derivative to calculate, unlike sigmoid or tanh. It’s either 1 or 0. For LReLU, either 1 or a small epsilon. You can feel the training difference when you run hundreds or thousands of epochs.

This gives ReLU a big advantage where numerical precision of your approximation doesn’t matter. It is hard to approximate smooth, continuous functions with ReLU, because it is a piecewise linear function. A mental image: try to approximate a polynomial (smooth continuous function) using a piecewise linear function. It’s okay (if you use many pieces), but approximate. sigmoid/tanh are smooth, so they are well suited in this case. But when the exact approximation is not the main issue, ReLU is an good enough and much faster.

For example in a computer vision network, you don’t care whether it the intermediate value in your network is 0.2 or 0.21, but you care about having thousands or millions of neurons to train.

Not having a derivative to calculate means you don’t have a “vanishing gradient” problem anymore (i.e. being far from the origin, where you derivative is quasi-zero and your weights update too slowly). However ReLU has an alternative feature: it is known as the “dying ReLU” problem. When the ReLU neuron evaluates a value below zero, the gradient is also zero. At some point, if the ReLU evaluates all the values to be zero, the neuron becomes useless. Sometimes it is not a problem, it creates sparsity and the network and makes it faster to run. But when you train the network for a long time, a big chunk of your network becomes inactive and unable to learn new features, and it becomes a problem. Leaky ReLU doesn’t have this problem.

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

Ans. 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,

Name three ways you can produce a sparse model.

Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)?

Ans. 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.