3.2 Lesson: Advanced Training Techniques

Exploding and Vanishing Gradients

As neural networks get deeper, training becomes more difficult — not necessarily because the model is too complex but because of what happens during backpropagation. In particular, gradients can become too small to be useful (*vanishing gradients*) or too large to be stable (*exploding gradients*). Both issues can seriously slow down or derail training.

What Causes the Problem?
Backpropagation works by repeatedly applying the chain rule to compute gradients from the output layer back to the input. In deep networks, this means multiplying many small or large numbers together — one for each layer. This can go wrong in two different ways:

- **Vanishing Gradients:** If the weights or activations are small (e.g., < 1), the gradients can shrink exponentially so that early layers learn very slowly.
- **Exploding Gradients:** If the weights or activations are large (e.g., > 1), the gradients can grow exponentially, leading to wildly unstable updates.

What Are the Solutions?

- **Use ReLU or a variant**: The ReLU activation function is a popular choice in deep networks because it avoids saturation in the positive direction — unlike sigmoid or tanh — which helps preserve gradients during backpropagation and speeds up training. However, standard ReLU can suffer from the "dying ReLU" problem, where some neurons output zero for all inputs and stop learning. To address this, several **variants** have been developed:
  - **Leaky ReLU**: Allows a small, non-zero slope for negative inputs to keep gradients flowing.
  - **Parametric ReLU (PReLU)**: Learns the slope of the negative side during training.
  - **ELU/GELU**: Smooth alternatives that aim to combine the benefits of ReLU and tanh.
- These alternatives can improve performance and training stability, especially in very deep networks.

- **Add normalization layers**: Adding layers like **batch normalization** helps keep activations stable during training, reducing the risk of both exploding and vanishing gradients. These are simple to use in Keras and often improve training speed and reliability.
- Advanced techniques include **gradient clipping** and **weight initialization** strategies. For the most part, these will not be useful in this module.

Batch Normalization

As we learned previously, neural networks work best when the input samples are standardized to a mean of 0 and a standard deviation of 1. However, it can also be done **after hidden layers,** applied during the forward pass either to the mini-batch (batch normalization) or to the whole layer (layer normalization). Batch normalization is more common and is performed before the activation function is applied to the outputs:

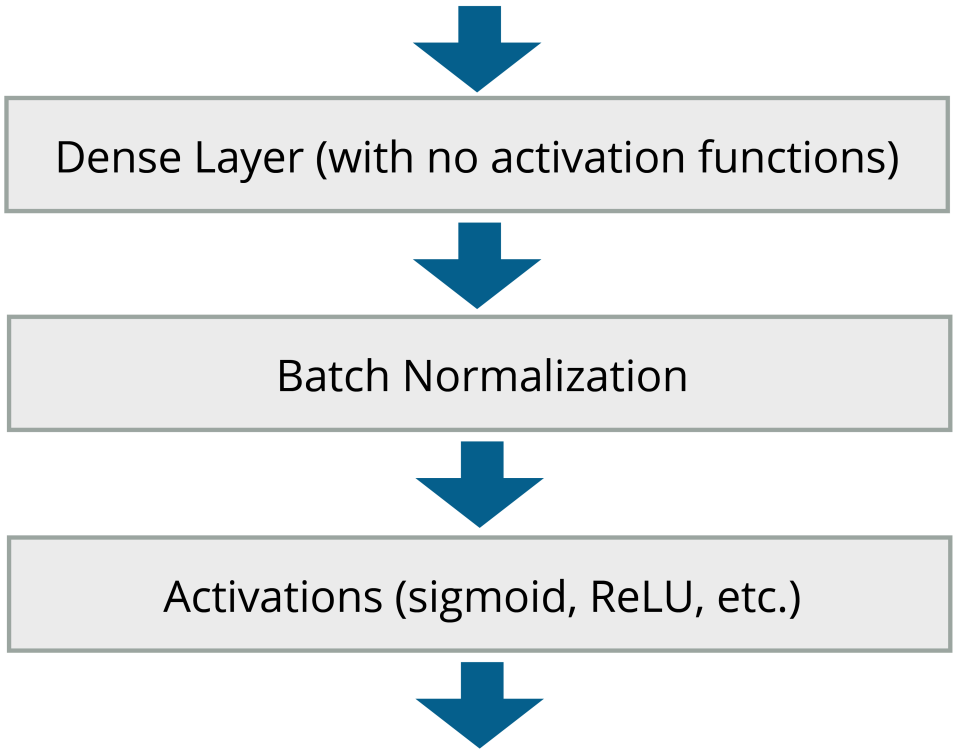

Figure 1. Batch Normalization Layer

Note carefully that batch normalization is applied **only to hidden layers**; standardization of the input is usually applied as part of dataset preprocessing, and the output layer for both regression and classification problems should never be normalized.

We will discuss using batch normalization in Keras in the Coding Video and notebook.

Preventing Overfitting Using Regularization and Dropout

Just as in linear models, we can apply regularization techniques that discourage the model from becoming overly complex and overfitting the training data.

L2 Regularization (Weight Decay)

In neural networks, **L2 regularization** — also known as **weight decay** — adds a penalty term to the loss function, just as in Ridge Regression:

$$L_{\text{total}} = L_{\text{data}} + \lambda \sum_i w_i^2$$

This penalty discourages large weight values, nudging the model toward smaller, more stable parameters. The strength of this regularization is controlled by the hyperparameter λ .

Smaller weights often lead to smoother, less overfit models that generalize better to new, unseen data.

Dropout

Deep networks have enormous capacity. While this is a strength, it can also be a liability — models can easily memorize training data, leading to overfitting. **Dropout** is a simple and surprisingly effective regularization technique that addresses this by adding randomness to the training process.

During training, dropout randomly sets a fraction of the units in a layer (either a hidden layer or the input layer) to zero on each forward pass. This prevents the network from becoming too reliant on any single neuron or feature, encouraging more robust representations. Here's how it works:

- **During training:** Each neuron has its output set to zero with probability p , meaning it *temporarily* doesn't contribute to the network's output for that training pass. The outputs of the remaining neurons are scaled up so that the sum of the activations from that layer are constant. Note that there is no permanent change to the "dropped" neurons.
- **During inference**: All neurons are used, and their activations are not scaled.

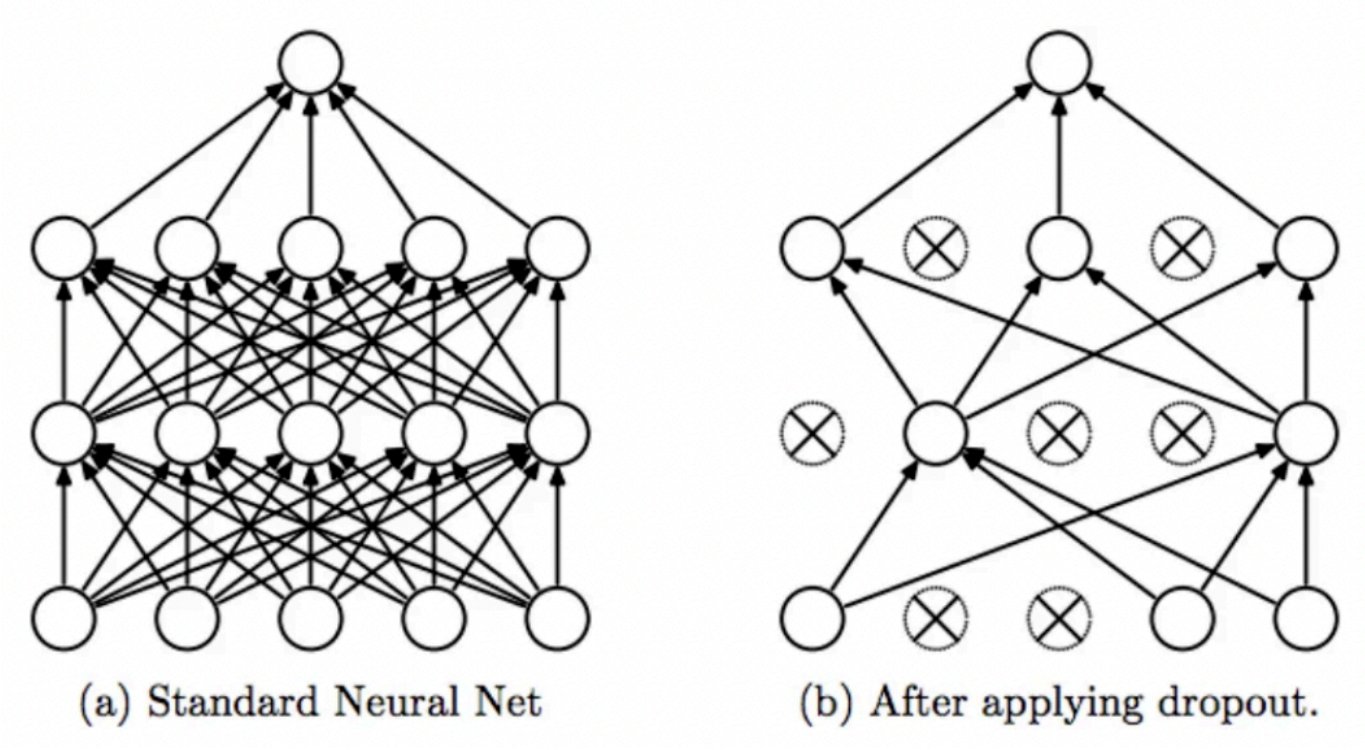

Figure 2: *Dropout in Neural Networks*

Source: Budhiraja, Amar. (2016, December 15). Dropout in (deep) machine learning (https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5). *Medium*.

Typical values for p :

- For hidden layers:
  - $p = 0.5$ is common in fully connected layers.
  - Sometimes we use $p = 0.2 ... 0.3$ if 0.5 is too aggressive or if the model underfits.
- For input layers:
  - Lower values are typically used, e.g., $p = 0.1 ... 0.2$, to avoid discarding too much raw information.

This simple trick has been shown to significantly reduce overfitting and improve generalization, especially in fully connected layers.

Conclusion

Training a neural network effectively requires more than just picking a model and feeding in data — it's about knowing what to watch, when to stop, and how to fix common training issues. In this lesson, we introduced the core techniques for monitoring training performance and improving generalization. In

the next lesson, we'll explore how to structure deep learning experiments, tune hyperparameters more systematically, and make informed choices when building real-world models.