First of all, when thinking about the architecture, simply check what worked in the past and take inspiration from the best performing models for tasks similar to yours. Of course consider also the amount of data and computational power that you have! If you don't have much of either, go for smaller models, especially at the beginning and then possibly iterate from there!

Model initialization:
- **Use suggested initializers** (Glorot, He, etc), unless you have a serious reason not to do so.
- **Transfer learning:** if you have few examples and a model which is able to deal with about the same data, you can transfer some or all the layers of the model already trained and fine tune only the last layer(s). (basically low level features detection might be useful for both tasks)
- **Unsupervised learning for supervised learning**: training a network through unsupervised learning can be beneficial as a starting point for supervised learning. This can be done through techniques of semi-supervised learning, GANs, autoencoders etc... The benefit of doing so is somewhat domain-specific: NLP tasks benefit greatly (e.g. learning embeddings before processing sentences), but computer vision not so much (unless labels are few), however it typically benefits a lot from transfer learning and tuning.
    - Unsupervised pre-training can have a regularizer effect and increase stability in convergence in training, reducing the variance of the estimation process of the weights, reducing overfitting.
    - Representations learned in deeper architecture are generally more effective/useful

Pre-processing:
- remember to **standardize / normalize** your input
    - this can mean different things in different domains, be sure to check what's appropriate.
- possibly **data augmentation** (mostly in computer vision)

LR schedule:
- Consider using **lr schedulers** (exponential decay, performance scheduling, ...)
    - Try many, different tasks respond differently to different scheduling.
- cosine annealing lr schedule is generally the go-to for transfer learning
- One can also reduce lr on plateaus (ReduceLROnPlateau) of a metric (valid loss)

Optimizers:
- SGD with momentum (and nesterov possibly) can converge to better results than RMSProp and adam, although more slowly.
- For long trainings/big models from scratch:

Need to reduce lr over time; initial lr is optimized by choosing a lr slightly larger (although not enough to cause severe instability) than the lr observed to be optimal in the first 100 iterations (but that wouldn't be in the complete training). Slowness in convergence in SGD is largely outweighted by the fast learning in the early phases; however, an idea can be to increase the batch size during the training process, to gradually improve convergence properties.
    - For really large models, if computational resources are available, asynchronous SGD can yield good performances much faster. Basicaly different machines share the view on the model's params and without much coordination they compute estimations of the gradient and update the params. The amount of improvment by step is clearly reduced (some machines can undo the work done by the other), but the learning process will be faster overall.
    - Note that, although not used in DL (to my knowledge), methods exist to reduce variance of SGD methods. Check SAG, SAGA and SVRG for more info.

- Adagrad reduces lr too fast and hardly converges to proper results (lr divided by sqrt of accumulated squared gradients); accumulation of the gradient of the WHOLE history of training is beneficial only in convex problems, for NNs it's not suggested.
- RMSprop is better than Adagrad since it uses an expon avg of the sq gradient, forgetting early gradients. Also momentum can help here.
- Adam uses both momentum and accumulated gradients as RMSprop but uses some bias correction to improve early learning. It's generally robust to hyperparams changes, but sometimes lr has to be adjusted anyway.

Loss:
- Task dependent, make sure you get it right

Activations:
Depend by the type of architecture but in general your default choice should be ReLU for your baseline. With RNNs or with exploding gradient problems tanh can be more appropriate. When iterating for better performances try ReLU variations to avoid dead neurons.

Number of neurons:
Make sure that your model is able to overfit to your data initially and then regularize. You want a model powerful enough to deal with the task and then restrain it, rather than having a model which can't deal with the task (underfits). HOWEVER, your model should be commesurate to your resources (both data and computational power). Make sure to print out the model architecture and understand where most of the parameters are. Are you sure your task require so many? Can you do something to reduce this number or make your network more efficient? In particular try to avoid huge FC layers (e.g. 2 FC layers with 2048 neurons are 4M parameters by themselves!!).

Regularization methods:
- l1, l2 regularization
- Dropout
- early stopping (should be used practically universally, free meal)
- low batch size
- batch norm (only slight effect due to noisy estimation of scale and shift params between batches)
- Noise in the input units, hidden, or even the target (label smoothing)
- Sparsity in the hidden representation, by L1/L2 penalty in the activations
- Bagging/ensemble (related to dropout)

Want to check if gradient is computed right?
- gradient checking (slow but effective)

Hyperparam optimization:
- Ideally don't use grid-search, use randomised search so you look at more values for every hyperparameter. Possibly zoom in on desired hyperparameters regions iteratively.
- Alternatively babysit your model, changing a single hyperparameter at a time.
- Consider using tools/libraries that automatically carry out hyperparams optimisation.

Ensembling:
- use multiple network (or even completely methods altogether, especially if they're independent) and average their outputs or use majority voting schemes.

Train, validation, test sets and data distributions:
- validation and test sets should come from the same distribution (which is the distribution you expect data to be generated in the future, when you have the model in production)

- The ratio to split the data into training/validation/test depends by how much data you have. e.g. for few data (thousands) 70/15/15 or 60/20/20, for a lot of data (millions instances) 98/1/1
- Training set can come from a different (but overlapping/related) distribution than the validation and test set, to include more data and help performance, when few data is available from the desired distribution (we only care about the metric on that though). It's better to include in the training set at least a decent number of occurrences from the desired distribution however.
    - Analysis of bias and variance should also be carried out more carefully, because the distributions of training and validation are different: e.g., there might be a large gap between training and validation loss but possibly most of the training data is composed of easier instances.
    - Splitting the training set into a further training-validation set (which is not used during training but only to assess performance) can be done to assess such problems. If there's a gap between training performance and training-validation performance there's indeed an overfitting problem. If there's not such a gap but there is wrt the validation set then it's only a data mismatch problem (and possibly error analysis by distribution could be beneficial for confirmation, i.e. see if the data included in the training set coming from the validation set distribution is indeed performing on par with the validation set).
    - Data mismatch problems can be alleviated by collecting more data coming from the validation distribution or in general making the training set more similar (e.g. artificial data synthesis once a problem is spotted; careful the data generated should be varied/not too repetitive).
- There could be a gap between validation and test performance, expecially when a lot of hyperparameter tuning is done, basically optimizing for the validation set involuntarily.
- Careful about splitting the data if multiple examples come from the same individual! The splitting should be done at individual level, not at example-level, since some information from examples included in the training set can strongly affect the performance on examples from the same individual in the validation/test set!
- Careful about hyperaparams optimization through several iterations! You might be unwillingly overfitting to the validation data and performing poorly on the test set!

Multi-task learning:
- we can have models with multiple outputs,possibly at different levels but not necessarily, which perform better than specialised models, because they're able to extract meaningful patterns from the data at different levels.

End-to-end or multi stage? (i.e. split the task in multiple easier sub-tasks or not?)
- basically this depends on the difficulty of the task and the amount of data available. If a lot of data (1M+) is available then end-to-end can be simpler and lead to better results.
- check the concept of shaping in AI.

Optimize a ML system, strategy, debugging:
- Error analysis: consider looking into which part of the system is most responsible for error and tackle that (e.g. clean up incorrect labels)
- Actually observe the output of the model and try to understand what went wrong, start from worst mistakes in terms of loss.
- Monitor histograms of activations and/or gradient. Also gradient norm can be helpful, especially relative to the magnitude of the params (i.e. the ratio between the two to see how fast learning is proceeding)
- Consider counting dead neurons if using relu (or switch to gelu, leaky relu)

Other tricks:
- Reducing the third dimension in convnets: the number of channels in convnets can be made flexible by using **1x1 convolutions**. This can be especially important to reduce the number of params in a network. Also 1x1 convolutions adds non-linearity to the convnet.
- **Gradient clipping**: sometimes, especially in RNNs, gradient can suddenly be large, and although it points in the right direction, if lr isn't reduced the parameters can move extremely far. This is avoided by reducing the step size when such large norms are detected.
- **Parameter sharing and tying**: if we know that some params should be close, we can induce it by adding a penalty, in the form of a L2 penalty of their difference, in the loss. When they should be equal instead we talk about parameter sharing, which can largely reduce the memory footprint (as it happens for CNNs).
- **Polyak averaging**: this is done for example in some deepRL algorithms.
- **Residual connections** have proven to be VERY helpful in many deep architectures.