Structure of the Net

We'll first introduce some notation. Let x_i be the i-th input and y_i be the i-th output (the final one). We denote the k-th output of layer j as $v_{j,k}$. $\alpha_{j,k,l}$ is the coefficient on output l of the previous layer $(j-1)$ in the k-th node of layer j. Let $\beta_{j,k}$ be the constant term for node k in layer j (these are often called 'bias' terms in ML settings). φ_j is the activation function for layer j. The activation function takes a scalar input and returns a scalar output. Common choices for the activation function are the sigmoid function $(1+\exp(-x))^{-1}$ and the ReLU function $\max\{0, x\}$. There are many other possible options out there for activation functions, however.

As a first example, suppose we have a network with N inputs and Q outputs with one hidden layer with M_1 nodes. Then, we can write the network as

$$v_{1,j} = \varphi_1\left(\beta_{1,j} + \sum_{k=1}^{N} \alpha_{1,j,k} x_k\right)$$
$$y_l = \beta_{2,l} + \sum_{j=1}^{M} \alpha_{2,l,j} v_{1,j}.$$

For a second example, consider if, instead, we had a network with two hidden layers with M_1 and M_2 nodes each. We can write this network as

$$v_{1,j} = \varphi_1\left(\beta_{1,j} + \sum_{l=1}^{N} \alpha_{1,j,l} x_l\right)$$
$$v_{2,k} = \varphi_2\left(\beta_{2,k} + \sum_{j=1}^{M_1} \alpha_{2,k,j} v_{1,j}\right)$$
$$y_m = \beta_{3,m} + \sum_{k=1}^{M_2} \alpha_{3,m,k} v_{2,k}.$$

We can continue this way if we want to add layers. When we talk about adding nodes, we talk about changing M in the first example or M_1 and/or M_2 in the second example. We can also transform the outputs suitably, depending on what restrictions we want to place on it. For example, if we want output m to only take on values between zero and one, we can apply the sigmoid function to it.

The structure of this model means that we can very easily compute outputs recursively. α_1, α_2, and α_3 are $M_1 \times N$, $M_2 \times M_1$ and $Q \times M_2$ matrices, while β_1, β_2, and β_3 are $M_1 \times 1$, $M_2 \times 1$, and $Q \times 1$ vectors. Then do the following:

$$v_1 = \varphi_1(\beta_1 + \alpha_1 x)$$
$$v_2 = \varphi_2(\beta_2 + \alpha_2 v_1)$$
$$y = \beta_3 + \alpha_3 v_2.$$

The fact that we're starting from the back and moving only in the forward direction makes this a feedforward network. I should also say that, in a slight abuse of notation, I put φ_i around vectors, when it's defined only for scalar inputs. That's just to say that it the function should be broadcast over the whole vector (applied to each entry). Ok, anyway, that's nice and easy.

Before moving on to training, we should make note of the fact that this function is quite amenable to differentiation; it welcomes it. Consider again the second example with two layers, we can quite easily compute $\frac{\partial y_m}{\partial x_l}$.

$$\begin{aligned}
\frac{\partial y_m}{\partial x_l} &= \sum_{k=1}^{M_2} \alpha_{3,m,k} \frac{\partial v_{2,k}}{\partial x_l} \\
\frac{\partial v_{2,k}}{\partial x_l} &= \varphi_2' \left(\beta_{2,k} + \sum_{j=1}^{M_1} \alpha_{2,k,j} v_{1,j} \right) \left(\sum_{j=1}^{M_1} \alpha_{2,k,j} \frac{\partial v_{1,j}}{\partial x_l} \right) \\
\frac{\partial v_{1,j}}{\partial x_l} &= \varphi_1' \left(\beta_{1,j} + \sum_{l=1}^{N} \alpha_{1,j,l} x_l \right) \alpha_{1,j,l}.
\end{aligned}$$

This is just straightforward chain rule stuff, but it leads to some ideas about how things can be implemented efficiently. We can rewrite this:

$$\frac{\partial y_m}{\partial x_l} = \nabla y_m \nabla v_2 \nabla_{x_l} v_1,$$

where v_1 and v_2 are $M_1 \times 1$ and $M_2 \times 1$ vectors. In the notation above, I define ∇y for an $n \times 1$ vector y with m inputs $\{x_i\}_{i=1}^m$ as the $n \times m$ matrix (i.e. the Jacobian)

$$\begin{bmatrix}
\frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_m} \\
\frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_m} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_m}
\end{bmatrix}.$$

I denote ∇y_i to be the i-th row of this matrix and $\nabla_{x_i} y$ to be the i-th column. So, we've reduced this problem to what's basically matrix multiplication since, when evaluated at a point (cheap and easy to do), ∇y_m, ∇v_2, and $\nabla_{x_l} v_1$ are $1 \times M_2$, $M_2 \times M_1$, and $M_1 \times 1$ matrices, respectively. Indeed, ∇y_m is just $\alpha_{3,m,\cdot}$. The others are similarly easy to evaluate, given the structure of each layer.

What's also nice about this way of looking at it is that it suggests an efficient way of computing it. We can operate recursively. Compute $d_0 = \nabla y_m$, then compute $d_1 = d_0 \nabla v_2$, and finally compute $d_2 = d_1 \nabla_{x_1} v_1$, which is a good way of going about this. It's so good that it, in fact, has a name. This is called backpropagation. It's sort of a mirror image of the feedforward procedure we did to compute the outputs in the first place.

Training of the Net

Let $\Phi(\mathbf{x};\Theta)$ be a neural network with $N \times 1$ input vector $\mathbf{x}$ and parameters Θ, which includes all the constant and coefficient terms for each layer. It's defined as in the previous part and we know how to evaluate it and its gradients efficiently. With all that in hand, we want to choose Θ to minimize some loss function $\mathcal{L}(\Theta)$, which is defined over this loss function. Usually, this is some type of non-linear least-squares thing, where we want to minimize

$$\mathcal{L}(\Theta) = N^{-1}\sum_{i=1}^{N}(y_i - \Phi(\mathbf{x}_i;\Theta))^2,$$

which has gradient and hessian

$$\nabla_\Theta\mathcal{L} = -N^{-1}\sum_{i=1}^{N}(y_i - \Phi(\mathbf{x}_i;\Theta))\nabla_\Theta\Phi(\mathbf{x}_i;\Theta)$$
$$\nabla^2_\Theta\mathcal{L} = -N^{-1}\sum_{i=1}^{N}\left((y_i - \Phi(\mathbf{x}_i;\Theta))\nabla^2_\Theta\Phi(\mathbf{x}_i;\Theta) - \nabla_\Theta\Phi(\mathbf{x}_i;\Theta)\nabla_\Theta\Phi(\mathbf{x}_i;\Theta)'\right),$$

where ∇_Θ applied to a scalar-valued function returns a column vector. Of course, this can also be any other loss function. The point is that the loss, the gradient, and the hessian are very easily computed by hand and/or by automatic differentiation packages.

At this point, we must be chomping at the bit to apply some sort of Newton method. It would have GREAT convergence properties because we are using the exact derivatives, rather than the numerical approximations. Not so fast. Sometimes Θ can be massively high-dimensional. When it gets on the order of a few thousand, the attractiveness of this option wears off and we have to resort to gradient-based methods. Of course, when Θ is more manageable (as I think it might be in quite a few cases of ours), we can still use it.

So, suppose that Θ is really high-dimensional. Maybe only computing and applying the gradient is feasible. In that case, we use gradient descent or some variation of it. This procedure updates the parameters as follows:

$$\Theta_{t+1} = \Theta_t - \gamma_t\nabla\mathcal{L}(\Theta_t),$$

where γ_t is called the learning rate. It can change with each step. Different methods have different ways of setting γ_t. The important thing is knowing that this is a common procedure. Other methods, instead of the pure gradient, may use some function of the gradient or a weighted average of past gradients. This is all done in the hopes that this algorithm may be better behaved. Rapidly changing search directions due to changes in the gradient along our path might make for slower convergence, so we dampen changes to the search direction. An example of a very popular algorithm that uses these approaches is ADAM (the name is derived from 'adaptive moment estimation'; further details on this method here).

Newton methods usually perform far better than pure gradient-based methods, which can be hard to tune and take a long time to converge. In practice, it's probably a good idea to prefer Newton methods whenever feasible. In cases where it's not feasible, ADAM is the state-of-the-art, standard optimizer for large-scale neural nets. It often dominates vanilla gradient descent methods, but on challenging problems, it's a good idea to apply different methods and see what works best from the point of view of the loss function.

Using the Net to Solve Models

For this, we're just going to think about the stochastic neoclassical growth model since that contains mostly all the principles we need to solve even bigger and more complicated models. We write it as

$$V(k,z) = \max_{k'} \left\{ u\left(zf(k) + (1-\delta)k - k'\right) + \beta \sum_{z'} P(z'|z)V(k',z') \right\}.$$

Approach 1: Only Approximate $V(k,z)$*.*

Let $\Gamma(k,z;\Theta)$ be a neural net approximating $V(k,z)$. If Γ is a good approximation of $V(k,z)$, we know that the envelope condition should hold. That is

$$\frac{\partial \Gamma}{\partial k}(k,z;\Theta) = \left(zf'(k) + 1 - \delta\right) u'\left(zf(k) + (1-\delta)k - k'\right).$$

This allows us to define a policy function $k' = \gamma(k,z;\Theta)$, since

$$k' = zf(k) + (1-\delta)k - [u']^{-1}\left(\left(zf'(k) + 1 - \delta\right)^{-1} \frac{\partial \Gamma}{\partial k}(k,z;\Theta)\right).$$

With automatic differentiation packages, we can compute $\frac{\partial \Gamma}{\partial k}$ exactly, which allows us to get an exact function for $\gamma(k,z;\Theta)$, given our approximation $\Gamma(k,z;\Theta)$. Given this policy function, we can define an error function. This error function is

$$\begin{aligned}\varepsilon^{\Gamma}(k,z;\Theta) = \Gamma(k,z;\Theta) &- u\left(zf(k) + (1-\delta)k - \gamma(k,z;\Theta)\right) \\ &- \beta \sum_{z'} P(z'|z)\Gamma(\gamma(k,z;\Theta),z';\Theta).\end{aligned}$$

Then define $\varepsilon(\Theta) = \{\varepsilon_i(\Theta)\}_{i=1}^{I} = \{\varepsilon^{\Gamma}(k_i,z_i;\Theta)\}_{i=1}^{I}$, where $\{(k_i,z_i)\}_{i=1}^{I}$ is some set of points that we got somehow. They could be obtained by random sampling. They could be obtained by forming a grid. They could also be obtained by simulating our current approximation of the model a number of times. There are a number of options available to us. Then, with our sequence of errors, our overall loss function $\mathcal{L}(\Theta) = L(\varepsilon(\Theta))$, where $L(\cdot)$ is some function that maps our errors to a scalar.

One particular example, which is what we typically use in practice, is

$$\mathcal{L}(\Theta) = I^{-1} \sum_{i=1}^{I} \varepsilon^{\Gamma}(k_i, z_i; \Theta)^2.$$

Then, we apply the same training techniques as usual to find Θs that minimize this loss function. These are the ones that generate a function that solves our problem.

What are the potential difficulties with this approach? First, if $\frac{\partial \Gamma}{\partial k}$ does not satisfy some conditions (e.g. not-positive or large enough in magnitude), we might not get a real, positive k'. So, our initial guess for Θ has to be such that $\Gamma(k, z; \Theta)$ satisfies all those conditions. Second, this will probably be a larger computational burden than the other approaches because we're performing lots of operations and the automatic differentiation package has to differentiate through all those. Third, this is possibly unstable. We could easily move into a part of the parameter space where $\gamma(k, z; \Theta)$ does not exist (like mentioned in the first part).

Approach 2: Only Approximate $g(k, z)$*.*

Let $\gamma(k, z; \Theta)$ be a neural net approximating $g(k, z)$. If γ is a good approximation of g, then it should satisfy the Euler equation

$$\begin{aligned}
&u'\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta)\right) \\
&= \beta \sum_{z'} \left[P(z'|z)\left(z'f'(\gamma(k, z; \Theta)) + (1-\delta)\right)\right. \\
&\times \left. u'\left(zf(\gamma(k, z; \Theta)) + (1-\delta)\gamma(k, z; \Theta) - \gamma(\gamma(k, z; \Theta), z'; \Theta)\right)\right].
\end{aligned}$$

As before, this also implies an error function.

$$\begin{aligned}
\varepsilon^{\gamma}(k, z; \Theta) = \beta \sum_{z'} &\left[P(z'|z)\left(z'f'(\gamma(k, z; \Theta)) + (1-\delta)\right)\right. \\
&\times \left. u'\left(zf(\gamma(k, z; \Theta)) + (1-\delta)\gamma(k, z; \Theta) - \gamma(\gamma(k, z; \Theta), z'; \Theta)\right)\right] \\
&- u'\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta)\right).
\end{aligned}$$

We could also consider

$$\widetilde{\varepsilon}^{\gamma}(k, z; \Theta) = \frac{\varepsilon^{\gamma}(k, z; \Theta)}{u'\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta)\right)},$$

which can sometimes be more stable. We can define the loss function and train it following steps we've already outlined.

This seems simpler than the previous method and we can define a $\gamma(k, z; \Theta)$ that automatically satisfies the budget constraint by taking

$$\gamma(k, z; \Theta) = (zf(k) + (1-\delta)k)\Psi(k, z; \Theta),$$

where $\Psi(k, z; \Theta)$ is a neural network with one output that only takes on values between zero and one. We don't have to make any modifications to how we train the parameters of this model.

So, what's the problem? Well, in principle, it's possible for the neural net to learn a solution that solves the Euler equation, but violates the transversality condition. There is nothing in our approach here to rule that out. We want to be aware of that issue and make sure that the transversality condition is not violated. The previous approach does not have this problem. The transversality condition will be satisfied if our approximation is a value function that satisfies the associated Bellman equation.

Approach 3: Approximate $V(k, z)$ *and* $g(k, z)$

In this approach, we approximate $V(k, z)$ with a network $\Gamma(k, z; \Theta_\Gamma)$ and approximate $g(k, z)$ with a network $\gamma(k, z; \Theta_\gamma)$. If these are good approximations, they must satisfy the Bellman equation and the envelope condition. That is:

$$\Gamma(k, z; \Theta_\Gamma) = u\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta_\gamma)\right) + \beta \sum_{z'} P(z'|z)\Gamma(\gamma(k, z; \Theta_\gamma), z; \Theta_\Gamma)$$

$$\frac{\partial \Gamma}{\partial k}(k, z; \Theta_\Gamma) = \left(zf'(k) + (1-\delta)\right) u'\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta_\gamma)\right).$$

For each of these, we can define our error functions

$$\begin{aligned}\varepsilon^\Gamma(k, z; \Theta_\Gamma, \Theta_\gamma) &= \Gamma(k, z; \Theta_\Gamma) - u\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta_\gamma)\right) \\ &\quad - \beta \sum_{z'} P(z'|z)\Gamma(\gamma(k, z; \Theta_\gamma), z; \Theta_\Gamma) \\ \varepsilon^\gamma(k, z; \Theta_\Gamma, \Theta_\gamma) &= \frac{\partial \Gamma}{\partial k}(k, z; \Theta_\Gamma) - \left(zf'(k) + (1-\delta)\right) u'\left(zf(k) + (1-\delta)k - \gamma(k, z; \Theta_\gamma)\right),\end{aligned}$$

which we can stack into vector-valued error function

$$\varepsilon(k, z; \Theta_\Gamma, \Theta_\gamma) = [\varepsilon^\Gamma(k, z; \Theta_\Gamma, \Theta_\gamma), \varepsilon^\gamma(k, z; \Theta_\Gamma, \Theta_\gamma)]',$$

which we can plug into a loss function so that we have some function of $\{\Theta_\Gamma, \Theta_\gamma\}$ to feed into the optimizer. This is again another problem that can be dealt with in the framework we talked about earlier. Once we can reframe our economic problem in terms of loss functions, then solving the model becomes a purely numerical issue.

The upside of this method is that it circumvents the difficulties we saw with the earlier methods. We don't have the same problems with finding sensible starting values for Θ_Γ or have the same level of concern about numerical instability. We also don't have to worry about violations of the transversality condition. However, both of these benefits come with additional computational costs. We have to perform more operations and fit more parameters.

An Aside on the Endogenous Gridpoint Method

As inputs, we specify a grid for capital (N_k points) and productivity (N_z points). We also have a transition matrix q, where $q_{ij} = P(z' = z_j | z = z_i)$. At step t of our procedure $g_{ij,t}$ is our conjectured choice for k′ given state (k_i, z_j). To start the procedure, we initialize $g_{ij,0}$, which is an $N_k \times N_z$ matrix. Let us call cash on hand w and let $\mathcal{I}(x_{out}, x_{in}, y_{in})$ be an interpolation function. The inputs are vectors and the output y_{out} is a vector of identical dimension to x_{out}. Pick a tolerance ε and a maximum number of iterations T. Then, for each t, we apply the following steps

1. For each $i \in \{1, \ldots, N_k\}$ and $j \in \{1, \ldots, N_z\}$, compute

$$\widetilde{w}_{ij} = [u']^{-1} \left(\beta \sum_{k=1}^{N_z} q_{jk}(z_k f'(k_i) + (1-\delta)) u' \left(z_k f(k_i) + (1-\delta)k_i - g_{ik,t} \right) \right) + k_i$$
$$w_{ij} = z_j f(k_i) + (1-\delta)k_i.$$

2. For each $j \in \{1, \ldots, N_z\}$, compute

$$g_{\cdot j,t+1} = \mathcal{I}(w_{\cdot j}, \widetilde{w}_{\cdot j}, k_\cdot)$$

   and for each i, j, set $g_{ij,t+1} = \max\{g_{ij,t+1}, k_0\}$;

3. Compute

$$\Delta_{t+1} = \max_{ij} |g_{ij,t+1} - g_{ij,t+1}|;$$

4. If $\Delta_{t+1} < \varepsilon$ or $t + 1 \geq T$, return $g_{ij,t+1}$. Else, set $t = t + 1$, $g_{ij,t} = g_{ij,t+1}$ and continue.

The idea here is basically backward induction. If we knew we were going to follow a certain policy tomorrow, what would our policy be today? We continue this backward induction until we our policy tomorrow is essentially the same as it is today (i.e. our algorithm converges). This algorithm is VERY efficient. It requires only interpolation, rather than root-finding.