CS 536 : Deep Learning without Pictures - a Terrible Idea 16:198:536

1 What is a Neural Network? Why is a Neural Network?

Previously, we've discussed classification / regression models of the form

$$f(\underline{x}) = \sigma(w_0 + w_1x_1 + w_2x_2 + \ldots + w_kx_k), \tag{1}$$

where the output of the model was some function of a linear function of the data. This was the case with perceptrons/SVMs using the sign function, this is the case for logistic regression using the sigmoid function, and this is the case for linear regression using the identity $\sigma(z) = z$.

While we've seen a lot of value in these models, there is a lingering sense of dissatisfaction - why only linear functions of the raw data components? Surely there are more relevant features or aspects of the data! We saw with the **xor** data for instance that for data of the form (x_1, x_2), correct classification needed to look at features like x_1^2, x_2^2, x_1x_2. What we would really like potentially is to look at models of the form

$$f(\underline{x}) = \sigma(w_0 + w_1h_1(\underline{x}) + w_2h_2(\underline{x}) + \ldots + w_{k'}h_{k'}(\underline{x})), \tag{2}$$

where each h_i represents some relevant computed feature of the data $\underline{x}$.

But what features? How do we know what 'relevant' aspects of the data should be computed as part of the classification/regression model?

The historical solution to this problem is to construct them by hand - through some kind of background knowledge of the problem or the data, construct combinations of the data components that seem relevant to the issue at hand, and once these features have been computed, all the previous discussions on regression and model fitting can be applied.

But this results in either a) pulling feature functions 'off the shelf' (see Kernel Methods for SVMs) that may or may not be applicable to the data, or b) basing your solution off your own intuition and understanding of the model. In either case, what we would really prefer is some notion of features that are data driven / data defined. The data itself should tell us what features are relevant to the problem.

This suggests the following solution: let h_i itself be a parameterized function, such that we can learn the 'ideal' parameters of the function from the data itself through model fitting. So we might have for instance,

$$\begin{aligned} f(x_1, x_2) &= \sigma(\alpha_1 + \alpha_2h_1(x_1, x_2) + \alpha_3h_2(x_1, x_2)) \\ h_1(x_1, x_2) &= \sigma(\alpha_4 + \alpha_5x_1 + \alpha_6x_2) \\ h_2(x_1, x_2) &= \sigma(\alpha_7 + \alpha_8x_1 + \alpha_9x_2) \end{aligned} \tag{3}$$

which gives a 'total' model of:

$$f(x_1, x_2) = \sigma(\alpha_1 + \alpha_2\sigma(\alpha_4 + \alpha_5x_1 + \alpha_6x_2) + \alpha_3\sigma(\alpha_7 + \alpha_8x_1 + \alpha_9x_2)). \tag{4}$$

This is a highly non-linear parametric function in nine parameters, and we could consider trying to find values for all 9 parameters that fit this model to the data. In the case of the **xor** data and σ as the sign function, we can actually get a correct classifier using the above model.

But why stop here? Why must the features we compute be functions of linear combinations of the data? We could nest further still - the output could be a function of features of features of features of data. Eventually we must stop - eventually we must compute *something* about the raw data itself. But by nesting these models sufficiently deeply

and considering sufficiently many features at every level, we can in fact build parametric models that can model arbitrarily complex data to very high degrees of accuracy. These models are neural networks.

A Theorem on Neural Networks: Consider a data set of the form $\{\underline{x}^i, y^i\}_{i=1,\ldots,m}$. We can model this data to an arbitrary degree of accuracy with a model of the form

$$f(\underline{x}) = w_0 + w_1 h_1(\underline{x}) + \ldots + w_K h_K(\underline{x}), \tag{5}$$

where each $h_j(\underline{x}) = \text{sigmoid}(w_0^j + \underline{w}^j.\underline{x})$, for appropriate values of weight vectors and biases and number of features K. That is, *any data set can be modeled arbitrarily accurately with a neural network with a single hidden layer that is sufficiently fat.*

The proof of this is involved, but notice that $\text{sigmoid}(z) - \text{sigmoid}(z+\delta)$ for small values of δ creates a function that is close to zero everywhere except for a small bump centered at $-\delta/2$ (the algebra for this is involved but doable). The proof then observes that *any* real function can be approximated by adding together and scaling sufficiently many bump functions located in different places.

2 Basic Design and Structure

Instead of writing out an enormous nested function, it is convenient to organize neural networks into *layers* of *nodes*. Each node takes a weighted combination of the nodes below it in the network, and outputs a non-linear function of this linear combination. Let $\underline{\text{out}}^t$ be the vector of outputs of nodes at level t - with the understanding that $\text{out}_0^t = 1$ at every level t, introducing the bias term. The nodes at the 0-th level represent the input to the network, so $\underline{\text{out}}^0 = \underline{x}$. The final output, at level K, is the final output of the system $f(\underline{x}) = \underline{\text{out}}^K$. *It is convenient to ignore the 0-th component bias term on the final level.* The 'links' between the levels are that for every $t = 1, \ldots, K$, and any node j on level t,

$$\text{out}_j^t = \sigma(\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1}), \tag{6}$$

where σ is some non-linear 'activation function', and $\underline{w}^{t-1}(j)$ represents a vector of weights on the inputs to node out_j^t.

Note: Depending on the kind of output from the network we want (for instance in a regression network versus a classification network) it may be convenient to take a different activation function on the final level, to ensure that the values of $\underline{out}^K$ match the structure we need. In regression problems, it is common to omit the activation function on this last level and just output a linear combination of the features below.

At this point, there are a number of basic questions to ask about building and utilizing neural networks:

- How many layers should it have?
- How many nodes should each layer have?
- What activation function should be used?
- How should the error or loss of the model be assessed?
- How can the model be trained to minimize this error?

The number of layers basically determines the level of complexity of the features being considered - the more layers, the more opportunity there is for the raw data components to be built up into more complex features of features of features. The more nodes per layer there are, the more information can be pulled in to inform the construction of new features. In general, these two architecture choices are relatively arbitrary and determined via trial and error. The larger the network is generally, the more expressive and flexible it can be - but this additionally risks overfitting. Additionally, while anything can be modeled by a sufficiently fat network, this may require a huge number of parameters. We can frequently trade with for depth, and get an as expressive network with much fewer parameters that need training.

The question of activation function is an interesting one. Historically, common activation functions have been

$$\begin{aligned} \text{sigmoid } \sigma(z) &= \frac{1}{1+e^{-z}} \\ \tanh \sigma(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}}. \end{aligned} \tag{7}$$

Both of these share the feature that for very large values of z (large positive or large negative) the function is relatively flat, and there is a linear section of the function near $z = 0$.

But more modern architectures have focused on the following kinds of activation functions:

$$\begin{aligned} \text{rectified linear unit (ReLU) } \sigma(z) &= \max(0, z) \\ \text{leaky linear unit } \sigma(z) &= \max \alpha z, z, \alpha \approx 0.1. \\ \text{exponential linear unit } \sigma(z) &= \begin{cases} z & \text{if } z \geq 0 \\ \alpha(e^z - 1) & \text{if } z < 0. \end{cases} \end{aligned} \tag{8}$$

Each of these have the behavior of linear growth for positive z, and almost flat behavior for negative z. The value of the second two will come from the training algorithms we discuss, that will rely on the derivatives of the various activation functions to inform how to modify the parameters. For the ReLU activation function, because it is constant 0 for negative inputs, the derivative is a constant and therefore relatively uninformative on how the input to the function should be tweaked (and parameters tuned) to improve performance.

This describes the architecture of the model generally. But to use neural networks, we need to be able to assess how good the model is (a notion of loss or error) and we then need to be able to tweak the model to try to reduce the loss or error (training). As usual, we exchange philosophical notions of learning for more concrete notions of mathematical optimization.

But the error of a model depends on the problem being posed. Regression error differs from classification error. The typical error for regression problems is the *squared error loss*,

$$L = \sum_{i=1}^{m} ||f(\underline{x}^i) - \underline{y}^i||^2. \tag{9}$$

Note here, we are utilizing the flexibility of neural network structure to consider multi-dimensional outputs and regression problems as well.

For classification problems, error is typically assessed in terms of number of misclassifications. That is,

$$L = \sum_{i=1}^{m} \mathbb{I}\{f(\underline{x}^i) \neq y^i\}, \tag{10}$$

where y^i is the class of data point $\underline{x}^i$, and the output of the model f is taken to be a predicted class value. The problem with this generally is that this loss function is discrete valued, and not continuous as a function of the

parameters of the model f. As a result, we cannot rely on our usual optimization algorithms like gradient descent that depend on the derivatives. However, as discussed in the appendix, we could consider (for binary classification problems) an alternative formulation in the following way: let $f(\underline{x})$ model *the probability* that $\underline{x}$ is in class 1, and therefore $1 - f(\underline{x})$ models the probability that it is not in class 1, or rather is in class 0. We can define a logistic error loss as

$$L = \sum_{i=1}^{m} - \left[y^i \ln f(\underline{x}^i) + (1 - y^i) \ln \left(1 - f(\underline{x}^i)\right)\right]. \tag{11}$$

See the appendix for a more detailed discussion. This loss function is a smooth function of the parameters of the model, so the usual algorithms can be applied to try to minimize loss.

We can generalize classification to multi-class classification in two ways: one, if there are multiple classes, y^i could indicate which class $\underline{x}^i$ belongs to, and if the possible classes are $1, \ldots, C$, the above formula generalizes to

$$L = \sum_{i=1}^{m} - \left[\sum_{c=1}^{C} \mathbb{I}\{y^i = c\} \ln f_c(\underline{x}^i) \right], \tag{12}$$

where f_c represents the result of 'output node c', and predicts the probability of being in class c. One important thing to observe is is that if the classes are exclusive, the probabilities of belonging to each of the classes needs to sum to 1. In this case, it is common to apply a 'softmax' function at the final output layer, so if that

$$\text{out}_c^K = \frac{e^{\text{out}_c^{K-1}}}{\sum_{c'=1}^{C} e^{\text{out}_{c'}^{K-1}}}. \tag{13}$$

In this case, we get that the output nodes are all positive, and in fact sum to 1 and therefore can be taken as probabilities.

Note that if instead of y^i representing the class value, the vector $\underline{y}^i$ is a vector of 0s for ever non-applicable class, and 1 for the applicable class, then the above can be described more concisely with the formula

$$L = \sum_{i=1}^{m} - \left[\sum_{c=1}^{C} y_c^i \ln f_c(\underline{x}^i) \right]. \tag{14}$$

This is known as the cross entropy loss, and actually generalizes in the following way: suppose that y_c^i was did not indicate whether data point i was in class c but rather a probability of being in class c (for instance the result of some kind of polling). The above loss can also be applied to try to get the predicted probabilities of f_c to match the 'true' probabilities of y_c^i.

This leaves the question of training - once the network has been designed, how can we produce parameter values to try to minimize the above loss? As indicated, if the loss function is a continuous, differentiable function of the parameters, then we can try to do some form of gradient descent to reach a minimum of the loss. While the loss function for neural networks will typically be wildly complicated and far from convex, the layer by layer structure of the network ensures that the gradients can actually be computed very easily.

3 Training: Backpropagation

The basis for most neural network training algorithms is gradient descent. That is, if W represents the full collection of all weights in the model, update the collection according to

$$W_{\text{new}} = W_{\text{old}} - \alpha \nabla_{W_{old}} L, \tag{15}$$

where $\nabla_{W_{old}} L$ is the collection of derivatives of the loss with respect to each of the weights, and α is some constant step size. This moves the old model in the direction of a new model with a smaller value of the loss. Iterating this, with some qualifications on α, should converge to a local minimum of L, and give a model that tries to minimize the loss over all.

For a number of reasons, it is frequently useful to consider *stochastic gradient descent* - instead of doing gradient descent with respect to the full loss L, we only do gradient descent on the loss with respect to one of the data points, $L_i = L(\underline{x}^i, \underline{y}^i)$ for some data point, chosen differently each iteration of the algorithm. The reason for this is two- if not three-fold. First, it is computationally cheaper to compute the gradients with respect to one of these loss terms rather than every loss term in the full sum for L. Additionally, it can be shown that gradient descent is fairly 'robust' so that even if you are only approximating the full gradient with the partial gradient in this way, SGD approximates GD in its convergence. Further, SGD can actually be less likely to get trapped in local minima, because it is less sensitive to fine grained details in L - it doesn't see the full function L at any point and so it can't see local minima to get trapped in them.

A sort of midpoint between gradient descent and stochastic gradient descent is *minibatch gradient descent* where instead of computing gradients with respect to the full loss function, or only the loss on a single data point, the loss is treated over some subset or batch of data in each iteration. This provides more information for the descent step and therefore should have better convergence, but still provides some of the computational savings and robustness of SGD.

In any of these three cases, the algorithm depends on the ability to calculate derivatives of L with respect to any of the weights. That is, we need to be able to compute

$$\frac{\partial L}{\partial w_{i,j}^{t-1}}, \tag{16}$$

for any $t = 1, \ldots, K$, and i as any node in level $t-1$, j as any node in level t (aside from the constant/bias node $j = 0$).

On its face, this is a complicated prospect - but the structure of the network makes it quite easy. Consider the question - how does the weight $w_{i,j}^t$ contribute to the loss over all? It only contributes to the loss through the value of out_j^t. By the chain rule, we have that

$$\frac{\partial L}{\partial w_{i,j}^{t-1}} = \frac{\partial L}{\partial \text{out}_j^t} \frac{\partial \text{out}_j^t}{\partial w_{i,j}^{t-1}} \tag{17}$$

Given the between layer relationship: $\text{out}_j^t = \sigma(\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1})$,, we have that

$$\frac{\partial \text{out}_j^t}{\partial w_{i,j}^{t-1}} = \sigma'\left(\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1}\right) \frac{\partial}{\partial w_{i,j}^{t-1}} \left[\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1}\right] = \sigma'\left(\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1}\right) \text{out}_i^{t-1}, \tag{18}$$

hence

$$\frac{\partial L}{\partial w_{i,j}^{t-1}} = \frac{\partial L}{\partial \text{out}_j^t} \sigma'\left(\underline{w}^{t-1}(j).\underline{\text{out}}^{t-1}\right) \text{out}_i^{t-1}. \tag{19}$$

So to complete the computation of the derivative with respect to the weight, it suffices to compute the derivative with respect to the output node value.

For $t = K$, the final output layer, the derivative of loss with respect to out_j^K is computed fairly directly, but depends on the specifics of the loss function. For instance, if $L = ||\underline{\text{out}}^K - \underline{y}||^2$, we get $\partial L / \partial \text{out}_j^K = 2(\text{out}_j^K - y_j)$. The derivatives at this level are easy, but loss function dependent.

For $t = 0, \ldots, K-1$, we can ask the question, how does the value out_j^t contribute to the final loss? We have that out_j^t contributes to every node at level $t+1$, so that by the chain rule we have that

$$\frac{\partial L}{\partial \text{out}_j^t} = \sum_k \frac{\partial L}{\partial \text{out}_k^{t+1}} \frac{\partial \text{out}_k^{t+1}}{\partial \text{out}_j^t}. \tag{20}$$

We can unpack this second term again using the link between levels: $\text{out}_k^{t+1} = \sigma(\underline{w}^t(k).\underline{\text{out}}^t)$, giving

$$\frac{\partial \text{out}_k^{t+1}}{\partial \text{out}_j^t} = \sigma'\left(\underline{w}^t(k).\underline{\text{out}}^t\right) \frac{\partial}{\partial \text{out}_j^t}\left[\underline{w}^t(k).\underline{\text{out}}^t\right] = \sigma'\left(\underline{w}^t(k).\underline{\text{out}}^t\right) w_{j,k}^t. \tag{21}$$

Substituting, we have that

$$\frac{\partial L}{\partial \text{out}_j^t} = \sum_k \frac{\partial L}{\partial \text{out}_k^{t+1}} \sigma'\left(\underline{w}^t(k).\underline{\text{out}}^t\right) w_{j,k}^t. \tag{22}$$

Hence from the above, we see that the derivatives at layer t can be computed in terms of the derivatives at layer $t+1$. This is the backpropagation effect - derivatives at the last layer $t = K$ are computed based on whatever the loss function is, then those derivatives are propagated backwards to compute derivatives at the next layer, then the next, tracking back all the way to the input layer. Tying this all together, we get the backpropagation algorithm:

The Backpropagation Algorithm: The derivatives of Loss with respect to any of the weights can be computed, layer by layer, as follows:

- At $t = K$, compute
$$\Delta_j^K = \frac{\partial L}{\partial \text{out}_j^K}, \tag{23}$$
for each output node j, based on the specific loss function.
- For $t < K$ (in decreasing order), compute
$$\Delta_j^t = \frac{\partial L}{\partial \text{out}_j^t} = \sum_k \Delta_k^{t+1} \sigma'\left(\underline{w}^t(k).\underline{\text{out}}^t\right) w_{j,k}^t. \tag{24}$$
- Any weight can then be updated according to
$$\left(\text{new } w_{i,j}^t\right) = w_{i,j}^t - \alpha \frac{\partial L}{\partial w_{i,j}^t} = w_{i,j}^t - \alpha \Delta_j^{t+1} \sigma'\left(\underline{w}^t(j).\underline{\text{out}}^t\right) \text{out}_i^t. \tag{25}$$

Author's Note: This is predicated on the idea that every node is computing σ of some linear combination of the nodes below it. If this is not true (for instance the last layer doing something different, in the case of regression networks or softmax classification networks), the update equations must be modified accordingly. The above represents a general guideline for you to modify as necessary for your specific models.

3.1 Initialization

Backpropagation and related update methods provide a mechanism for updating an existing model to a (hopefully) better model. But where to begin? What initial values should you take for your weights?

A natural answer is to initialize all weights to zero. The problem with this however is that it induces an unhelpful symmetry in all of the nodes, as they are all computing the exact same thing. This symmetry can be very hard to break, and will keep the weights from converging to anything useful.

An alternate approach is *random initialization* - pick an initially random model and go from there. Typical choices include normal initialization - initializing all weights to be normally distributed with mean zero and some small variance. This provides two benefits: one, alternate models and be produced by re-initializing the weights at random and running training again. If you are unhappy with the current performance of your model, you can reset the weights and start over. This is frequently used in training neural networks, replicating the initialization and training a number of times and taking the best model that results.

Additionally, centering the weights on zero is a good idea for a number of reasons. Consider the sigmoid and tanh activation functions, which are very flat away from $z = 0$, and very responsive in the neighborhood of $z = 0$. In this case, if weights are initialized to be far from zero, this may put a node in these 'unresponsive regions' where the output of the activation function is flat (and therefore the derivatives are flat). In this region, a lot of tweaking of the parameters will do very little to change the response of the node, and we see generally that the network will learn very slowly. Keeping weights near zero helps to ensure that the initial network is in this highly responsive region where node outputs are sensitive to changes in the weights, and learning can happen quickly.

However, it is not enough to just center the weights at zero. Consider a situation like

$$y = w_0 + w_1 x_1 + \ldots + w_{100} x_{100}. \tag{26}$$

If all the w_i were selected to be normal with mean zero an variance 1, then the final value of y (depending on the specifics of the x values) could have a variance on the order of 100. This could easily put y into the 'flat' or unresponsive regions of the activation function. We see then that it is additionally important to make sure that the weights are clustered tightly around zero, so that the variance of the linear combination is small as well.

This leads to ideas such has Xavier Initialization and Hu Initialization - in these schemes, weight initialization needs to have smaller variance the more inputs there are to a given node. So that if $w_{i,j}^t$ is a weight going into node j, if j has In_j input nodes (frequently referred to as the 'fan in', we would want

$$\text{var}(w_{i,j}^t) = 1/\text{In}_j. \tag{27}$$

Other schemes exist but all are variants of this same idea (some balancing variance of not only the input, but also the output, to try to make backpropagation more effective). Xavier Initialization was originally given with

$$\text{var}(w_{i,j}^t) = \frac{1}{\frac{1}{2}(\text{In}_j + \text{Out}_j)}. \tag{28}$$

The above is typically applied to normal random initializations - other distributions such as the uniform distribution can be considered. Xavier Initialization applied to uniform distributions typically takes the form of

$$w_{i,j}^t \sim \text{Unif}\left[-\sqrt{\frac{6}{\text{In}_j + \text{Out}_j}}, +\sqrt{\frac{6}{\text{In}_j + \text{Out}_j}}\right], \tag{29}$$

which produces the same variance.

3.2 Preventing Overfitting

Neural Networks from one perspective, as discussed here, are nothing more than highly parameterized non linear functions that are incredibly expressive. There are two potential pitfalls to be aware of here:

- You need a lot of data to determine good values for the parameters.

- It is very easy to overfit to your data, to learn your training data too well and fail to generalize outside of it.

In many ways these are the same kinds of problems we've seen before with other models, and similar solutions can be applied here:

- **Regularization:** By adding a cost for the complexity of the model, this can help avoid overfitting to the data. Ridge and Lasso regression can easily be applied to neural networks, just augmenting the loss function with the appropriate regularization term. Note additionally that Lasso will have the added benefit of potentially pruning (by setting weights to 0) edges between nodes that don't need to be connected.
- **Early Termination:** By tracking the performance of the model on a testing or validation set over the course of training, we can terminate training when the error on the test set starts to increase (or at least stops decreasing). This is usually very effective.

Other things that we can potentially do for neural networks include *brain surgery* - given a trained neural network, it may be worth considering, can the network be pruned without sacrificing performance? This is connected to weight regularization, but we consider modifying the network itself. This can be very important to consider since sufficiently large networks can model anything - there is a lot of value to be had in the smallest possible network that still describes the data well.

4 A Biological Justification

So far we have justified neural networks as a model to consider just from generalizing our previous models to more computationally flexible models. This is fine, but we can also justify neural networks from the idea of attempting to model how brains actually work. Hence the name, 'neural' networks.

As an example, consider the visual processing done by neurons between the eye and the visual cortex. Light enters the eye and falls on a layer of photoreceptors in the retina. These are simple neurons that become more electrically active the more light that falls on them. These neurons are connected to another layer of neurons, further back in the retina. For these neurons, the more signal coming into them, the more active they become - and they are wired to the initial photoreceptors in such a way that their 'activity' tracks with the presence or detection of simple features, in this case edges, or lines of various orientations.

But there is another layer of neurons behind this, taking as 'input' the signal from these edge-detection neurons. The more active the input neurons are, the more active these neurons will be. These are wired to the edge detection neurons in such a way that they detect various combinations of edges in various orientations - otherwise known as shapes.

Layer by layer, each layer of neurons becomes 'active' if it is fed sufficient signal from its input neurons, and then it passes its own signal farther into the brain. These 'features of features of features' detecting neurons have been discovered, refined to such a degree that there are neurons that become active when the subject is looking at something as specific as a certain person's face.

This is the kind of structure that our artificial neural networks are attempting to mimic. One encouraging observation to make here is that when deep neural networks have been trained for vision or image recognition tasks, analysis of the nodes after the fact has revealed almost identical patterns of detection - edges, shapes, combinations of shapes, building up to complex features. This suggests that this pattern of feature structures is really quite effective - discovered simultaneously by billions of years of evolution, and artificially by training these mathematical models with gradient descent.

This to some extent is also the source of the interest in the sigmoid function as an activation function: neurons typically have a threshold of activation - if their inputs fail to meet that threshold, the neuron stays inactive. If the neuron's inputs rise above that threshold, the neuron 'fires', passing its signal up the network. However, neurons are also 'capped' in the sense that they are largely on or off, with some transition between these two states. This is effectively what we see with the sigmoid function - 'off' for inputs that are too small, and 'on' for inputs that are above that threshold.

5 Modern Approaches: Deep Learning

Neural networks are an old idea, dating back to the '50s. There has been a resurgence over the last 15 years or so, driven primarily by the following trends:

- Huge amounts of data, which are needed to train the many parameters of a neural network.
- Improved hardware, for crunching the considerable computations involved in training.
- Improved design choices for network architecture.

One of the big design choices that have made a lot of difference is the realization that while sigmoid functions are 'enough' for an activation function, there are actually computationally much better choices. In particular, the use of rectified linear units and their related functions has dramatically improved both the training and performance of networks generally.

Other advances have focused on network structure. In 'classic' neural networks, we consider dense connections between the layers, where every node is connected to every other node. Additionally, each layer is effectively 'flat', a vector of possible inputs. Both these things are not applicable to images, for instance - there is an inherent 2D structure to the input, so the input layer should take advantage of this geometry. Additionally, in processing images, features are frequently 'local': pixels located near each other are more strongly related than pixels that are located very far apart. Importing this 'locality' into the structure of the network can facilitate learning, because the network isn't trying to learn correlations and features between arbitrarily separated pixels.

Three big advances in network architecture are the following:

- **Convolutional Layers:** Convolutional layers express two ideas. The first is locality - that features may be locally defined, so that there is no point in having distant connections between nodes. Typically, convolutional layers have some inherent 'window' (with some width and length in the case of looking at images) and only nodes within this window serve as input to a given node in the next layer. The second idea is spatial invariance - if a feature is relevant in one location, it may be relevant in another location as well. It doesn't make sense to train one node to detect faces in a specific area, and another node, separately, to detect faces in a different area; both are trying to learn how to detect faces. If a feature is defined by its weights, convolutional layers implement this by replicating weights.

  In 1-D, you might get something that looks like this:

  $$f(x_1, \ldots, x_5) = \sigma(w_0 + w_1\sigma(a + bx_1 + cx_2 + dx_3) + w_2\sigma(a + bx_2 + cx_3 + dx_4) + w_3\sigma(a + bx_3 + cx_4 + dx_5)). \quad (30)$$

  Hence at the next level, we are computing the same feature on (x_1, x_2, x_3) as we are on (x_2, x_3, x_4) and (x_3, x_4, x_5). This would be a kernel or filter window of size 3, stride 1. However, there may not be a single feature that is most useful on a given window, so you typically specify the number of kernels or filters to

compute. In this way, on a 28x28 pixel image, a convolutional layer with a window of size 3x3, stride 1, and 10 different filters might generate a block of size 26x26x10 as input to the next layer. Note, replicating weights and forcing the network to be sparse in this way (to enforce locality) also serves as a form of regularization, cutting down on the number of available parameters and the overall complexity of the model - this helps avoid overfitting.

It is worth commenting here on training: because we want to replicate weights in various points, this modifies the backpropagation algorithm. But for a given weight w, we can still ask the question - how does this weight contribute to the total loss? If that weight modifies the input to node out_a and out_b for instance, then by the chain rule we have

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \text{out}_a}\frac{\partial \text{out}_a}{\partial w} + \frac{\partial L}{\partial \text{out}_b}\frac{\partial \text{out}_b}{\partial w}, \tag{31}$$

and the backpropagation equations can be modified accordingly.

- **Max Pooling Layers:** Max pooling layers also address this question of locality - in some specified window (as in convolutional layers), the output of a max pooling node is simply the maximum value from its input nodes. This serves a couple of different purposes. One, again it maps down the complexity of the model and serves as a kind of regularizer. Two, it can be viewed as introducing a sort of rotational invariance of features. Any rearrangement of the incoming feature values will have the same maximum value output

  Max pooling layers are conceptually straightforward, but they do add an extra wrinkle in terms of training, as now the derivatives are going to depend on which of the input nodes at a given node provided the maximum value.

- **Dropout Layers:** Dropout approximates the idea of training multiple models and pooling their predictions into one model. During training, each time the model is to be updated, we 'drop' some fraction p of nodes from the model - in other words we set the outputs that they pass to the next level to be zero. What this means is that during this step of training, the network is forced to try to learn *without the information those nodes would have provided*. This forces the network to get the most out of the information that is available at every step, and develop better models for the connections and correlations between certain features. Each time, this is done with a different fraction of nodes 'dropped out'. This can be seen as simulating a smaller network with fewer nodes within the main network. In addition to simulating training multiple models on the data, this can also be seen as modeling the idea of 'occlusion' - for instance in image recognition, the object of interest might be only partial visible or covered, such as a person wearing a hat that obscures part of their face. We would like our model to be able to handle situations like this, and make the most of the information that actually is available - training with dropout can be seen as simulating this.

  It's important to note that dropout is something that should only be done in *training*. During testing or in use, a network should always make use of all the data that is available to it. One computational implication of this is the following: if dropout on a given layer is done with probability p, during training the next layer was trained using an input that is effectively only a fraction p of what it would've been without dropout (if only half the nodes are giving input, the total input is half of what it would otherwise be). This means that during testing or evaluation, when nodes aren't being dropped, the input the next layer is getting is 'too big' by a factor of $1/p$ (if all the nodes are now giving input, the total input would be twice what the next layer is used to). It is convenient then to downscale the outputs of dropout nodes during testing/evaluation, shrinking their outputs by a factor of p, and putting the input of the next layer on the scale it is used to.

6 Applications

A A Digression on Classification

In typical classification models, we take $f(\underline{x})$ to be the predicted class (either class 0 or class 1) that the data point $\underline{x}$ belongs to, and we assess the loss of the model in terms of the number of misclassified data points. The problem with this generally is that loss is now a discrete function, and cannot be easily optimized with respect to the parameters of the model.

One way around this is to introduce a softening: let $f(\underline{x})$ be a real value between 0 and 1, and interpret this as the probability that $\underline{x}$ belongs to class 1 (so that $1 - f(\underline{x})$ is the probability that it belongs to class 0. Given a data set with 'correct' class values $y^i = 0$ or $y^i = 1$, we can asses 'the likelihood of these class values occurring for this data', or

$$\text{lik}(f) = \prod_{i=1}^{m} f(\underline{x}^i)^{y^i} (1 - f(\underline{x}^i))^{1-y^i}. \tag{32}$$

From a maximum likelihood perspective, we can consider the problem of trying to find the model f that maximizes this likelihood, to make the output data $\{y^i\}$ as likely as possible. However, maximizing products in this way is frequently difficult, so traditionally we take the log to simplify this and turn the product into a sum:

$$\begin{aligned} \ln \text{lik}(f) &= \sum_{i=1}^{m} \ln \left[f(\underline{x}^i)^{y^i} (1 - f(\underline{x}^i))^{1-y^i} \right] \\ &= \sum_{i=1}^{m} \left[y^i \ln f(\underline{x}^i) + (1 - y^i) \ln \left(1 - f(\underline{x}^i)\right) \right]. \end{aligned} \tag{33}$$

Therefore instead of trying to maximize the likelihood, we can consider the problem of maximizing the log likelihood. Or equivalently, minimizing the negative log likelihood, which gives us a new notion of loss for classification problems:

$$L = \sum_{i=1}^{m} - \left[y^i \ln f(\underline{x}^i) + (1 - y^i) \ln \left(1 - f(\underline{x}^i)\right) \right]. \tag{34}$$

This gives us a loss function that will be continuous/differentiable in terms of the parameters of the model f, and we can apply our usual training methods.