# 6.867: Lectures 9, 10: Neural Nets, Deep learning

## Contents

*These lecture notes have not been subject to careful scrutiny usually reserved for research papers.*

**Beta-version (caveat emptor)! Released early due to impending exam!**

# 1 Introduction

In previous lectures we have seen nonlinear classification, via explicit and **prespecified** choice of nonlinear feature maps or via implicit nonlinear maps using some fixed kernel function. A general guideline to selecting the nonlinear map or the kernel is to roughly tie it to what kind of prior knowledge we wish to model or impose. When working with nonlinear maps, one is led to wonder about the following question:

*What if we learn the nonlinear map from data instead of prespecifying it?*

As good MLers, you should often ask this type of question in a variety of settings!

**Motivating viewpoint:** Recall that we using kernels, for any point $x$ we have the implicit feature $\phi(x)$. Then we compute the prediction as

$$w^T \phi(x) + b = \sum_i \alpha_i y^{(i)} k(x^{(i)}, x). \tag{1}$$

Introducing a new feature map

$$\psi(x) := [y^{(1)} k(x^{(1)}, x), \ldots, y^{(n)} k(x^{(n)}, x)],$$

we can rewrite (1) as

$$w^T \phi(x) + b = \alpha^T \psi(x) + b.$$

In other words, we view the prediction (1) as being parametrized by $\alpha$, and then computed by using nonlinear features $\psi$. These features are not explicitly fixed in advance but they depend on the training data since $\psi$ depends on similarity to each of the $x^{(i)}$.

However, this $\psi$ is not "learned" from the data, in the sense that we do not really construct $\psi$ to optimize classification performance. The $\alpha$ vector is what we learn (e.g., via an SVM).

Ignoring crossvalidation for setting hyperparameters

Thus, if we could somehow also learn $\psi$ from the data, while optimizing classification performance, we may potentially obtain a more power classifier. However, this task seems enormously more difficult computationally. We present in these lectures one such approach, namely, *neural networks*, where we jointly learn the features and the classifier.

# 2 Feedforward neural networks

- A neural network (NN) contains several simple computational units called neurons (see Fig. 1).
- These units carry out simple computations such as linear classification, detecting maxima, computing linear transformations, etc.
- In its simplest setup, the network is arranged in layers. Raw input flows in, gets processed in a sequence of increasingly sophisticated levels (layers), until the final output is obtained.

## Structure

- **Input layer:** This is really just a synonym for the input data. Here the units store the coordinates of the input vectors. The input layer performs no computation.

We assume for simplicity that the inputs are just vectors. Later we'll allow matrices (images) too.

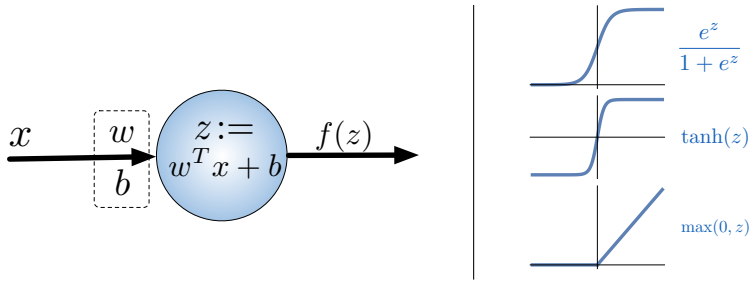



Figure 1: A single neuron / unit. The input $x$ flows in; $w, b$ are parameters of the neuron, which computes $w^T x + b$ and then transforms this value by "passing it through" a nonlinear function $f$. The output $f(z)$ is also called the "activation." The right part of the figure shows three different activation functions: sigmoid, tanh, and rectifier.

- **Hidden layers:** These layers perform complex transformations of the input signal. Think of these layers as generating increasingly complex features. In NN training, the art is in designing the *architecture* of the hidden layers, though as research progresses, this design is becoming somewhat less difficult.
- **Output layer:** This may be a single (or multiple) neuron (neurons), for instance a linear classifier (like SVM), or a softmax function (when working with multiple classes). This layer takes as input the features generated by the network, i.e., the output of the penultimate layer, and generates a prediction.

**Example.** Figure 2 illustrates a schematic of a neural network with hidden layers. The input layer "receives" $x \in \mathbb{R}^d$, and has a unit corresponding to each coordinate. This layer just copies its input, and thus the activations of the input layer are given by $a^1 = x$. The final layer's output is denoted by $F(x; \theta)$, i.e., it depends on the training data point $x$ and $\theta$ refers to the parameters of the entire network. This network has a single output unit, and we can have for example $F(x; \theta) = z$, which denotes the identity map (*cf.* Fig. 1).

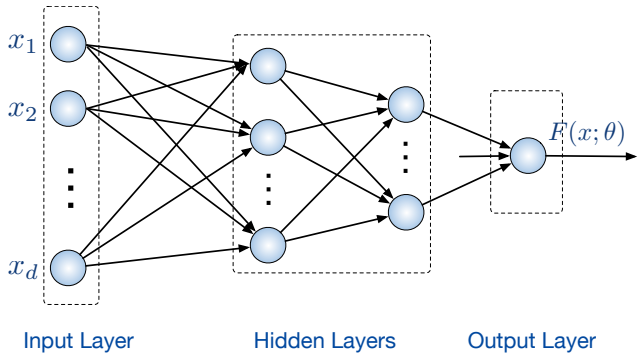



Figure 2: Illustration of a neural network with hidden layers

**Toy example.** Consider the trivial neural network in Fig. 3 that has just an input and an output layer but no hidden layers. This network just aggregates its inputs and computes the output $z = w^T x + b$. The network is parameterized by $\theta = (w, b)$. This network is nothing but a linear classifier: we can take the output $F(x; \theta) = z = w^T x + b$ and pass it through $\text{sgn}(\cdot)$ or any other function that we want to arrive at a linear classification decision (e.g., we could use $F$ within a loss function, when trying to optimize the network parameters).

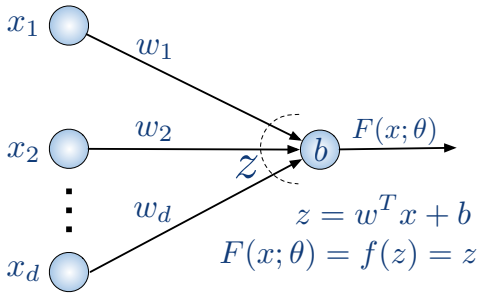



Figure 3: Illustration of a neural network with no hidden layers.

## 2.1 Hidden layers, learning representations

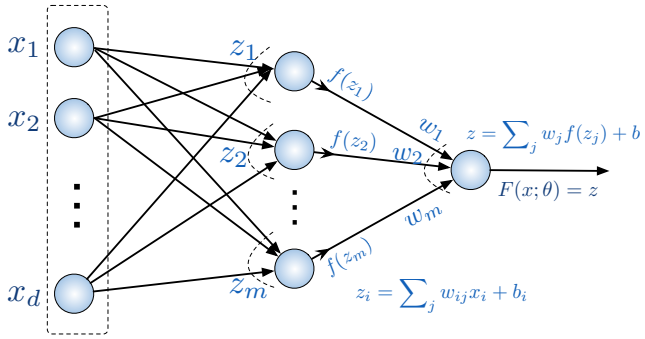



Figure 4: A neural network with 1-hidden layer.

Fig. 4 illustrates a neural network with 1 hidden layer. Hidden unit $i$ (for $1 \leqslant i \leqslant m$) computes its weighted input $z_i = \sum_j w_{ij} x_i + b_i$ and then outputs a feature $f(z_i)$. In other words, each unit generates its output in two steps:

$$\textbf{Aggregate:} \quad z_i = \sum_{j=1}^{d} w_{ij} x_i + b_i, \quad 1 \leqslant i \leqslant m,$$
$$\textbf{Activate:} \quad a_i = f(z_i).$$

Henceforth, we will often assume that $f(z)$ is the **rectifier**, i.e., $f(z) = \max(0, z)$. A neuron that employs the rectifier as its activation function is called a Rectified Linear Unit (ReLU).

In colloquial NN speech, it also common to refer to the rectifier function itself as the ReLU.

The output unit does not see the input data directly, but operates on the activations of the hidden units, and computes

$$z = \sum_{j=1}^{m} w_j f(z_j) + b$$
$$F(x;\theta) = z \qquad \text{(network output)}.$$

**Note:** The output unit's can be used as a linear classifier on (combined with a suitable activation function) the **new feature** representation $f(z_1(x), \ldots, f(z_m(x)))$ for an input data point $x$ (which can be a training or test data point). The network parameters are the weights and biases of each neuron.

**How powerful?** One may wonder how powerful is a neural network with a single hidden layer. After all each hidden layer unit is computationally as rich as linear classifier (its output has a nonlinearity of course). By putting together several such units and generating nonlinear features, how much can one gain? It turns out that (at least in theory), that a NN

with a single hidden layer is powerful enough to learn almost any continuous function. Cybenko's result, stated informally below, expresses this fact.

**Theorem (informal).** Let $f$ be a sigmoid (or similar) activation. Given any continuous function $h$ on a compact set in $\mathbb{R}^d$, there exists a NN with 1 hidden layer and a choice of parameters such that the output

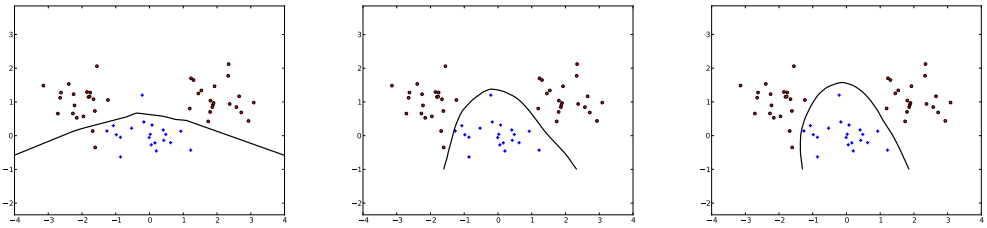
$$F(x) = \sum_i^N v_i f(w_i^T x + b_i)$$

approximates $h$ to any desired accuracy $\epsilon > 0$, i.e., $|F(x) - h(x)| < \epsilon$ for any $\epsilon > 0$.

There is an older result of Kolmogorov stating that if we are allowed to use different activation functions for each neuron, then we can even attain equality.

**Remarks:** The above result is interpreted as saying that a 1-hidden layer network is a *universal approximator*. However, this result, while theoretically elegant, should not lead to an overinterpretation. The number of neurons needed in the hidden layer may have to be exponentially large (in the input dimension). Moreover, even if the number turns out to be not so large, there is likely no tractable algorithm to learn the parameters needed to attain the approximation.

Figure 5 illustrates this point. It shows data that should be separable using a NN with 2 hidden units (why? *Hint:* look ahead at the next example). However, it is hard to learn the parameters of that network. Nevertheless, as we increase the network size, the number of parameter configurations that suffice to separate the data also greatly increases, and the chances of obtaining a classifier that manages to separate the data points increases. From the figure one also, however, already notices that as we increase the number of hidden units, we gain perfect separation and that the nonlinear classifier learned seems to overfit the data quite strongly.



10 units 100 units 500 units

Figure 5: Impact of increasing number of units in the hidden layer.

**Example.** Consider the nonlinearly separable points in Fig. 6 that we try to classify using a NN with 2 hidden units. Thus, we compute

$$\begin{aligned} z_1 &= w_{11}x_1 + w_{21}x_2 + b_1, \qquad \text{(1st unit)} \\ z_2 &= w_{12}x_1 + w_{22}x_2 + b_1, \qquad \text{(1st unit)} \\ f(z_1) &= \max(0, z_1), \qquad f(z_2) = \max(0, z_2). \end{aligned}$$

Thus, these units map each data point into new coordinates $x \mapsto [f(z_1), f(z_2)]$. To appreciate the power of these nonlinear features, notice how easily the points become separable (right plot in Fig. 6).

At the same time, we must observe that it can be challenging to learn the representation that works. Consider the normal vectors for the hyperplanes shown in the left plot. What would happen if we were to flip signs of these vectors? What feature vectors would we now learn? This example suggests that estimating parameters of a NN is not easy, and in general, with an increasing number of hidden neurons we can have an increasingly large

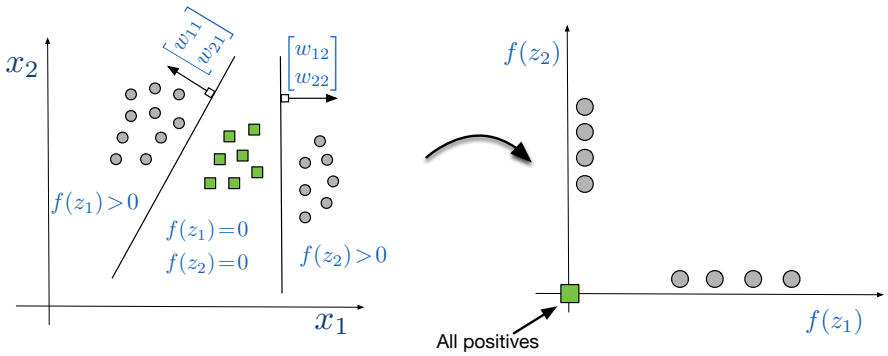



Figure 6: The two hyperplanes drawn correspond to the points where the outputs of the corresponding neurons are exactly zero (the normal vectors are drawn to show the region in space where the corresponding neuron's activation is positive). The gray circles denote negatively labeled points; the green squares denote positively labeled points. Observe how the points become linearly separable.

number of configurations, and not all may work. Nevertheless, as Fig. 5 shows, with large enough networks, we may be able to easily separate, though at the risk of overfitting.

## 2.2 Learning the parameters

We now consider the task of estimating the parameters of the NN given training data. As before, we set up a loss function that measures the fit to training data, and optimize it by selecting some loss function (hinge-loss, logistic loss, etc.) on the output generated by the network. Formally, we consider the empirical risk minimization problem:

$$\min_{\theta} \quad R_n(\theta) := \frac{1}{n}\sum_{i=1}^{n} \ell(y^{(i)}, F(x^{(i)};\theta)), \tag{2}$$

where $\ell(y,z) = \max(0, 1-yz)$ or $\ell(y,z) = \frac{1}{2}(y-z)^2$ or some other choice. In addition we can add a penalty term like $\lambda\|\theta\|^2$ or $\lambda\|\theta\|_1$ to regularize. Later on, we will comment on other methods for regularizing.

We minimize (2) by using SGD. Thus, at each iteration we pick a random training data pair $(x,y)$ and perform the update

More correctly, we attempt to minimize! This problem is non-convex and is typically intractable.

$$\theta \leftarrow \theta - \eta\frac{\partial \ell(y, F(x;\theta))}{\partial \theta}. \tag{3}$$

To implement (3) we must specify the stepsize $\eta$, as well we compute the stochastic gradient $\partial\ell/\partial\theta$ and define a procedure to initialize the iteration. We will comment on stepsizes and initialization later on in these notes; for now, let us focus on the task of computing the stochastic gradient. Thus, our aim is to compute $\partial\ell/\partial\theta$. For concreteness, consider a NN with a single hidden layer (as in Fig. 4) and with $\ell(y,z) = \max(0, 1-yz)$. Then, consider

$$\begin{aligned} z_j &= \sum_{i=1}^{d} w_{ij}x_i + b_j, \qquad \text{(input to jth unit)} \\ f(z_j) &= \max(0, z_j), \qquad \text{(output of jth unit)} \\ z &= \sum_{j=1}^{m} w_j f(z_j) + b \qquad \text{(input to output unit)} \\ f(z) &= F(x;\theta) = z \qquad \text{(NN output)} \end{aligned}$$

Let us focus on computation of $\partial \ell / \partial w_{ij}$. To aid this computation we appeal to the chain-rule of calculus, motivated by observing that a change to $w_{ij}$ changes $z_j$, which in turn changes $f(z_j)$, which eventually impacts $z$ and thus the loss $\ell$. Formally,

$$\begin{aligned}\frac{\partial \ell(y,z)}{\partial w_{ij}} &= \left[\frac{\partial z_i}{\partial w_{ij}}\right]\left[\frac{\partial f(z_j)}{\partial z_j}\right]\left[\frac{\partial z}{\partial f(z_j)}\right]\frac{\partial \ell}{\partial z} \\ &= [x_i][\![z_j > 0]\!][w_j]\begin{bmatrix} -y, & \text{if } \ell(y,z) > 0, \\ 0, & \text{otherwise.}\end{bmatrix}\end{aligned}$$

**Note:** A word about the above computation is in order. Actually, neither the hinge loss, nor the ReLU activation is differentiable. However, both are convex functions and are *subdifferentiable*. Formally, for a convex function $h$ the set of vectors

$$\{g \mid h(u) \geqslant h(v) + \langle g, u - v\rangle,\ \forall u\},$$

is called the subdifferential of $h$ at $v$ and is denoted by $\partial h(v)$. Any member of this set is called a *subgradient*. Importantly, if $v$ is a point at which $h$ is differentiable, the set $\partial h(v) = \{\nabla h(v)\}$, i.e., it is a singleton that agrees with the derivative. The ReLU $\max(0, z)$ is not differentiable at 0, otherwise it is differentiable. You should convince yourself that 0 is one of the subgradients of the ReLU — we can use **any** subgradient in the chain rule above, and for convenience we pick 0. This choice can have some other undesirable side-effects though, but we will retain it due to its simplicity.

The topic of subgradients of convex functions is treated more thoroughly in a class on convex optimization. If you are unfamiliar with this material, you should look it up.

## 3 Backpropagation

Suppose we have a more complex NN, e.g., a deep neural net with several hidden layers (e.g., Fig 2). How should we apply the chain rule? A change to a weight at the first hidden layer will impact the inputs and outputs of all the subsequent layers, and as such to apply the chain-rule we will have to aggregate the contribution from each neuron to the final output. This leads to a huge combinatorial explosion of the number of possibilities and it seems computationally intractable.

It was therefore, a remarkable achievement that a simple (in hindsight) method based on dynamic programming was discovered in the 1980s that bypasses this combinatorial explosion, by trading space for time, and ends up delivering a practical method to make application of the chain-rule computationally feasible.

This procedure is called *backpropagation*. The key idea behind it is similar to the motivation we used for the 1-hidden layer case described above. When theare are multiple layers, the (stochastic) gradients can be evaluated efficiently by propagating them backwards from the output (starting from the term $\partial \ell / \partial z$) to the inputs. To apply the chain rule, each previous layer much evaluate how the next layer affects the output. The process of using this intermediate information to be able to propagate the gradient backwards through the network is called backpropagation, or backprop for short.

We sketch the backprop procedure below. Imagine that our NN has layers $1, 2, \ldots, L$, where layer 1 is the input layer and $L$ is the output layer. Figure 7 sketches the quantities at play when transiting from layer $l-1$ to layer $l$ in the network.

To ease our derivations, we introduce vectorial notation. Thus,

$$\begin{aligned} a^l &:= (a^l_1, \ldots, a^l_{m'}), \qquad (m' \text{ units in layer } l) \\ z^l &:= (W^l)^T a^{l-1} + b^l \\ a^l &= f(z^l). \end{aligned}$$

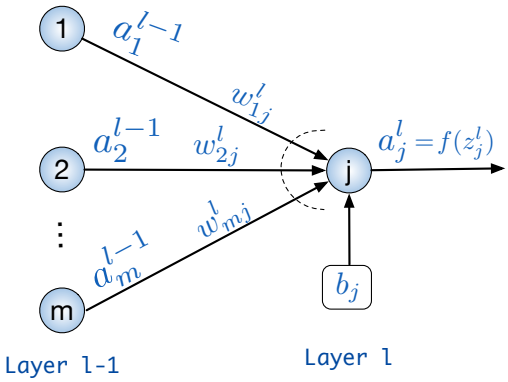



Figure 7: Connection of layer $l-1$ to neuron $j$ in layer $l$.

Here, $W^l$ denotes the weight matrix $[w_{ij}^l]$ (for $1 \leqslant i, j \leqslant m$); see Fig. 7. The vector $a^l$ denotes the activations of layer $l$, while $z^l$ denotes the vector of inputs, formed by combining the outputs of layer $l-1$ using the weight matrix $W^l$ and the vector of biases $b^l$.

We wish to compute $\partial \ell / \partial \theta$. Let us illustrate the details for $\partial \ell / \partial W^l$. Consider thus,

$$\frac{\partial \ell}{\partial W^l} = \frac{\partial z^l}{\partial W^l} \left[ \frac{\partial \ell}{\partial z^l} \right] = \frac{\partial z^l}{\partial W^l} (\delta^l)^T,$$

where $\delta^l := \frac{\partial \ell}{\partial z^l}$ denotes the "error" at layer $l$.

Observe that the first part $\partial z^l / \partial W^l = a^{l-1}$. Let us try to obtain a recursive formula for $\delta^l$. To that end, notice that when $z^l$ changes, it has an impact on $z^{l+1}$. So we write

$$\frac{\partial \ell}{\partial z^l} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \frac{\partial \ell}{\partial z^{l+1}} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \delta^{l+1}.$$

From the update equations we know that

$$z^{l+1} = (W^{l+1})^T a^l + b^{l+1} = (W^{l+1}) f(z^l) + b^{l+1}.$$

Differentiating wrt $z^l$, we thus obtain

$$\frac{\partial z^{l+1}}{\partial z^l} = \text{Diag}[f'(z^l)] W^{l+1},$$

where $\text{Diag}[f'(z^l)]$ denotes the diagonal matrix formed from the vector $f'(z^l)$. Thus, we obtain

$$\delta^l = \frac{\partial \ell}{\partial z^l} = \text{Diag}[f'(z^l)] W^{l+1} \delta^{l+1}. \tag{4}$$

Equation (4) contains $\delta^{l+1}$ on the right hand side, so we can inductively apply it to obtain

$$\delta^l = \text{Diag}[f'(z^l)] W^{l+1} \, \text{Diag}[f'(z^{l+1})] W^{l+2} \cdots W^L \delta^L.$$

Now observe that

$$\delta^L = \frac{\partial \ell}{\partial z^L} = \frac{\partial \ell}{\partial a^L} \frac{\partial a^L}{\partial z^L} = \text{Diag}[f'(z^L)] \frac{\partial \ell}{\partial a^L},$$

where the final derivative can be obtained easily since the loss is a simple function of the activation (output) of the final layer.

Thus, putting things together, we can summarize the backprop procedure as follows.

1. **Input:** Training pair $(x, y)$; set activations $a^1 = x$ for layer 1
2. **Feedfoward:** **for** each $l = 2, 3, \ldots, L$ compute:
$$z^l = (W^l)^T a^{l-1} + b^l$$
$$a^l = f(z^l)$$
3. **Output "error":** $\delta^L = \text{Diag}[f'(z^L)]\nabla_{a^L}\ell$
4. **Backpropagation:** **for each** $l = L-1, \ldots, 2$ compute:
$$\delta^l = \text{Diag}[f'(z^l)]W^{l+1}\delta^{l+1}$$
5. **Gradient:**
$$\frac{\partial \ell}{\partial W^l} = a^{l-1}(\delta^l)^T, \quad \frac{\partial \ell}{\partial b^l} = \delta^l.$$

This procedure computes for us $\partial \ell(y, F(x;\theta))/\partial\theta$ (where $\theta = (W^1, b^1, \ldots, W^L, b^L)$ is the vector of all the parameters). With this stochastic gradient in hand, we can finally perform the update (3).

# 4 Training a neural network

Even though we have now seen how to compute a stochastic gradient for a neural network, there are several other aspects of training a neural network that are worthy of noting.

- **Initialization:** Properly initializing a NN is very important. One of the main reasons is because the NN loss is a highly nonconvex function, so optimizing it to attain a "good" solution (i.e., a network that has good prediction ability) can be difficult and requires some careful tuning. When using ReLUs, (also because we chose to use zeros as subgradients at the points of non-differentiability), if we initialize the network weights to zero, all the gradients and activations will be zero, and will not change even after seeing training data. One choice is to initialize the weights randomly, chosen according to a Gaussian distribution with mean zero and variance $\sigma^2$ that depends on the number of neurons in a given layer (roughly to ensure that the random input to a unit in a layer does not depend on the number of inputs it receives, e.g., $d$ inputs for each neuron in a network with 1 hidden layer).
- **Unstable gradients:** Looking at the backpropagation rule, we see that there is a multiplication of a chain of matrices in there. If can easily happen that several of these matrices are "small" (i.e., norms $< 1$), so that when we multiply several of them, the gradient will decrease exponentially fast and tend to *vanish*. Conversely, if several of the weight matrices have large norm, the gradient will tend to *explode*. In both cases, the gradients are unstable. When training deep networks, coping with unstable gradients poses several challenges, and must be dealt with to achieve good results.
- **Activation functions:** The choice of activation function can be important while also affecting how we may initialize (e.g., when using sigmoids, to avoid saturation, and when using ReLUs to avoid zeros). Empirically, it has been widely reported that with ReLUs one typically obtains higher accuracy, though ReLUs are somewhat more sensitive to numerical issues. Also a problem known as "dying units" has been reported for ReLUs, where several units can get permanently clamped down at zero and never activated again. Several heuristics for dealing with this problem have also been suggested, and the practitioner should pay attention to these ideas.

- **Regularization:** The larger the neural network, the larger the number of parameters we have to learn for it, and therefore, the higher are the chances of overfitting. Indeed, overfitting is one of the most common problems of neural networks. Several methods for regularizing are worthy of consideration, e.g., increasing the size of training data (for instance by creating "virtual data", a classic ML idea); by reducing the step-size used for SGD; by initializing carefully; by using L1/L2-norm regularization (or other such penalties); by using dropout. The latter gained some popularity and offers a useful means for regularizing. We mention some more details below.

- **Loss function:** The choice of the loss function $\ell$ to use in (2) can be made according to the application at hand. Moreover, the nature of the output layer (namely, what activation functions it uses, and its size) is also tied to the task. For instance, if we are solving a regression problem, then we may prefer to use the square loss $\ell(y,z) = \frac{1}{2}(y-z)^2$ combined with linear activation for the output layer. If we are solving multiple independent binary classification problems (in this case we will have multiple neurons in the output layer) we can use the cross entropy loss with sigmoids as activation functions for the output layer; we can also use linear output combined with the hinge loss for binary classification. If we are solving a multiclass classification problem, then again we can use the multiclass version of cross entropy loss function combined with softmax activation.

## 4.1 Dropout

When fitting to the nitty-gritty (including noise) of the input, hidden units must rely on each other to co-adapt / complement the coverage of the data space. Hence, to hinder fitting to noise, we must avoid this co-adaptation. One approach is to randomly turn off units, say with probability 1/2, when training. Thus, for each data point, we randomly set the output of each hidden unit to zero. The neurons turned off are randomly chosen anew for each training data point, and also accounted for during backprop. For units that have been turned off for that round, the input weights coming in / activations going out are not updated, so the unit is effectively *dropped out* for that particular training sample. The neurons can thus rely on signals from their neighbors, only if a large number of them support these. This dropout strategy therefore provides a means to regularizing.

At test time we can simulate the impact of dropout in a simple manner. Since each neuron was on approximately only half the time during training, its contribution to the units in subsequent layers is just half of what it would be if it were on all the time. Thus, we simply multiply the outgoing weights of each neuron by 1/2 during test time.

There exist more detailed explanations of this idea of dropout, as well as theoretical interpretations. We omit those from our discussion, but encourage the interested reader to pursue the wider literature to learn more.

**Adversarial examples**$^\star$ We mention here briefly that the research field is actively seeking to find ways to regularize neural nets or to make them less sensitive to noise in the data. An interesting approach in this direction is by the explicit construction of "adversarial examples" that are designed to maximally mislead the network. Figure 8 shows an example drawn from a recent paper that was used to fool the NN into thinking that some (carefully crafted) random noise is an actual digit with very high confidence.

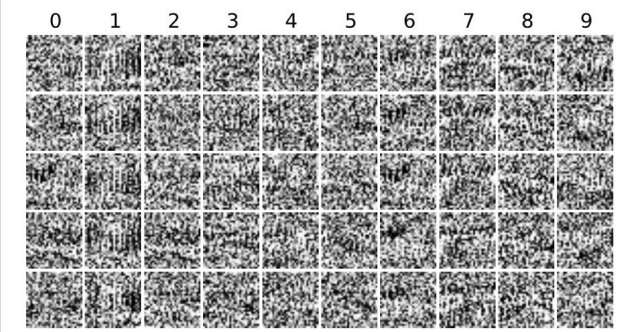



Figure 8: Adversarial example on MNIST. This figure is taken from a recent paper by Nguyen, Yosinski, Clune (CVPR 2015) that shows how to construct points that are clearly noise, yet a DNN believes with close to 100% confidence to be one of the digits 0-9.

# 5 Convolutional networks

We now move onto convolutional nets, one of the most successful NN architectures. The motivation is quite simple. Previously we assumed that the input data is a vector and then used fully connected (FC) layers, where each individual coordinate was connected to all the neurons in the hidden layer (and this pattern of FC layers was repeated throughout the network). However, suppose our input is not a vector but a 2D image. Then, treating the whole image as a vector seems to largely ignore the local structure present within the image. Nearby pixels are often "similar" or more closely related than far away ones. If we try to explicitly use this spatial structure, we could perhaps encode the input data better, which may eventually lead to higher classification performance. Indeed, if the aim were to recover local spatial structure, by merely using FC layers, we would need a gigantic amount of training data this structure.

This suggests that instead of mapping individual coordinates of the input layer to all neurons in the hidden layer, we should consider small groups of spatially close by coordinates and map them to the same neuron. Fig. 9 illustrates this idea.

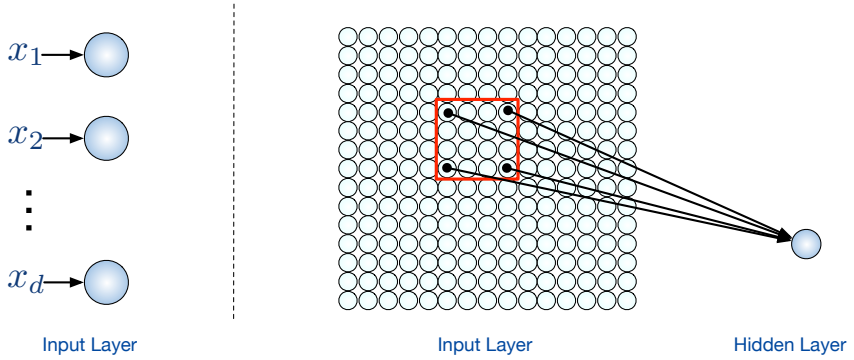



Figure 9: Connecting a patch in the input layer to a hidden neuron to use spatial structure. All the $4 \times 4$ input units (e.g., pixels) are connected to the same hidden unit.

In particular, we take different patches in the input layer and connect them to corresponding neurons in the hidden layer. The most common approach is to make a sliding window of patches, and each patch maps to its corresponding neuron in the hidden layer. Fig. 10 illustrates these connections.

As before we attach weights to each edge in the connections. Thus, for instance, in

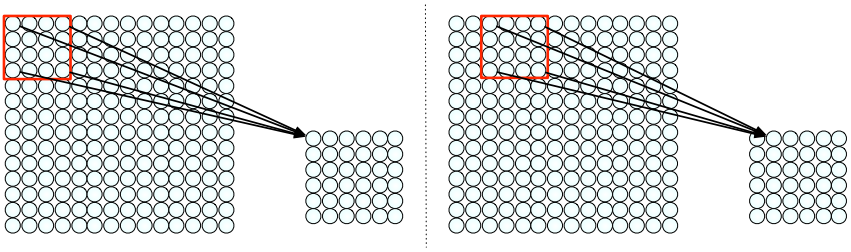


Figure 10: Connecting patches in the input layer to their corresponding hidden neurons. We are using $4 \times 4$ patches with a *stride* of 2, as can be seen from the image.

Fig. 10, each $4 \times 4$ patch is connected to a hidden neuron in the first hidden layers, requiring 16 weights for encoding it; each hidden unit also has a bias term. From the picture it is clear that we have 36 overlapping patches possible, which translates into a total of $16 \times 36 = 576$ weights in total to map the $14 \times 14$ input image (input layer) into a $6 \times 6$ hidden layer. If we were to use a stride of 1, then the number of hidden units would be 11, which translates into $121 \times 16 = 1936$ weights.

The above calculations indicate that as the input images get bigger, the number of weights required to map the input layer to the hidden layer grows very rapidly (try out a calculation for a $256 \times 256$ image to convince yourself). This explosion of weights leads to an "overparameterization" of the neural network, which is undesirable on various grounds, beyond just computation and storage demands.

The key behind convolutional nets, henceforth convnets or CNNs, is to use *weight sharing*, which just means that for each patch we will use the same weights! Thus, for all the 36 patches in Fig. 10, we use the same weight matrix, which requires just $4 \times 4 = 16$ weights, a dramatic savings.

What might be the function of a patch / hidden neuron in this case? Observe that now hidden unit $(p, q)$ will take the pixels in its patch as input and compute its activation

$$f\left(\sum_{i=1}^{4}\sum_{j=1}^{4} w_{ij}x_{i+p,j+q} + b\right)$$

i.e., the activation is computed by taking a weighted sum between the pixel values in a patch and the corresponding weights (which are independent of the patch). Thus, we may view this operation as extracting the same set of ***local features*** from each image patch.

## 5.1 The convolution operation

Figure 11 illustrates the above idea of extracting local features on an input image. Different choices of weights can be used to extract local features, such as edges, local peaks, etc., from each patch. An important point to note is that convolution is a linear operation. We move the same window of weights over all patches and compute linear combinations. Thus, if we were to view an input $d \times d$ images as a long vector (of length $d^2$), then the convolution operation could be represented by a matrix of size $d^2 \times d^2$, albeit a very sparse one.

**Exercise:** Suppose you convolve a $5 \times 5$ image with a $2 \times 2$ patch moved across with stride of 1. Write down the matrix representation of the convolution operation.

**Convolution is fast.** The above exercise shows that the convolution operation results in a highly structured matrix. The entire crux of convolution lies in not computing the operation by performing a dense matrix-vector multiplication (which, done naively, seems to require $O(d^4)$ operations). By exploiting the repetitive structure of the convolution, one can

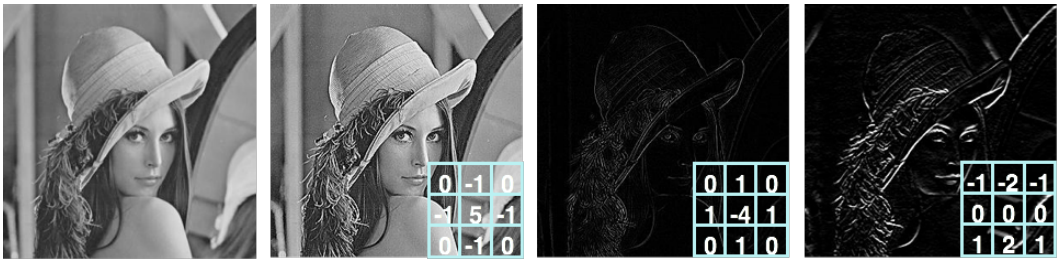



Figure 11: Examples of convolving the lena image with different filters. From left to right the filters are: sharpen, edge detect, strong edge detect (figure best viewed on screen).

resort to the Fast Fourier Transform (FFT) to perform the convolution in merely $O(d^2 \log d)$ operations, that is, in time essentially linear in the input size! This remarkable savings is computation is what makes convolution so appealing (and provides another reason for "weight sharing" across all patches).

## 5.2 Convnet architectures

Above we saw that convolution with patches helps us extract local features. When making connections between the input layer and the first hidden layer, we may prefer to learn several types of local features rather than just one set. In particular, we do not need to learn just one set of weights: we can learn several sets of weights, although as before, the way we use these weights is to continue being able to a fast convolution operation.

Figure 12 illustrates an example convnet architecture that operates on input images / matrices of size $28 \times 28$ and in the convolution layer, i.e., the first hidden layer, creates 3 feature maps by using patches of size $5 \times 5$ with a stride of 1. Therefore, we end up having $3 \times 24 \times 24$ neurons in the convolution layer.

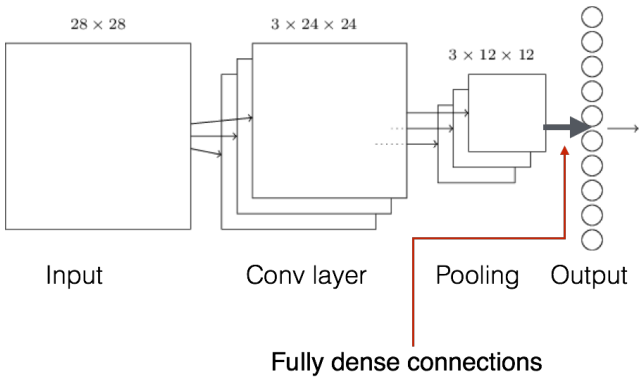



Figure 12: Example of a convnet architecture.

Before we close our discussion on convnets, we mention briefly another type of "convolution" layer that is often used with convnets, namely "pooling". The aim is to reduce the size and intermediate complexity of the layers somewhat by pooling together neurons that have similar activations. For instance in *max pooling* (illustrated in Fig. 13) we take patches of some size within the convolution layer, and apply a patchwise maximum when mapping the outputs of the conv layer to the next hidden layer, called the pooling layer. Clearly, other pooling / combining approaches are also possible (e.g., averaging). Importantly, each of the feature maps obtained in the conv layer is subjected to the same pooling,

and as illustrated in Fig. 12, this translates into the pooling layer having the same number of "slices" as in the conv layer.

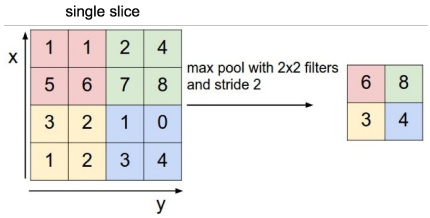



Figure 13: Max Pooling illustration

Finally, after a sequence of conv layers and pooling layers, we add a fully connected layer that is linked to all the neurons in the layer prior to it.

## 5.3 Perspectives, remarks

Convnets have driven a lot of the recent success of deep networks in computer vision tasks. Training them requires heavy computational resources and some care. However, they are not the final word on neural network architectures. Recently, deep network architectures based on a nonlinear computational flow graphs have become more popular due to their state-of-the-art performance on several benchmarks. Figure 14 illustrates one such architecture called the "residual network" that connects the input layer not only to the first hidden layer, but also directly to a deeper layer. We do not discuss these networks further, but encourage the interested reader to read through the links provided in the lecture slides.

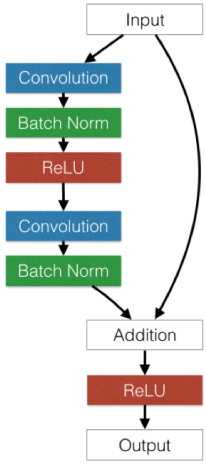



Figure 14: Residual Network Architecture (non-linear computational flow graph)